실시간 데이터베이스 시스템을 위한 효율적인 병행실행제어 알고리즘 설계☆

Design of an Efficient Concurrency Control Algorithms for Real-time Database Systems

이 석 재* 박 새 미** 강 태 호*** 유 재 수****
Seok-Jae Lee Sae-Mi Park Tae-ho Kang Jae-Soo Yoo

요 약

실시간 데이터베이스 시스템의 트랜잭션은 마감시간과 같은 시간 제약조건을 갖는다. 따라서 트랜잭션은 마감시간 내에 완료되어야 하며 동시에 데이터 일관성 제약조건을 만족해야 한다. 빠른 평균 응답시간 제공을 목표로 하는 기존 데이터베이스 시스템과 달리 실시간 데이터베이스 시스템은 트랜잭션의 마감시간 초과 비율과 마감시간을 초과한 트랜잭션에 의해 발생되는 비용에 의해 평가된다. 따라서 실시간 트랜잭션은 공정성과 시스템 처리율의 저하를 발생시킨다 하더라도 트랜잭션의 중요도 및 마감시간에 따라 스케줄링 되어야 하고, 항상 높은 우선 순위 트랜잭션의 선행 처리가 보장되어야 한다.

본 논문에서는 기존 실시간 스케줄링 알고리즘들이 갖는 문제점을 개선시킬 수 있는 새로운 스케줄링 알고리즘(*Multi-level EFDF*)과 펌, 소프트 실시간 트랜잭션을 고려한 병행실행제어 알고리즘(*2PL-FT*)을 제안한다. 또한 성능평가를 통해 *2PL-FT*와 *AVCC* 방법에 대한 트랜잭션의 재시작 비율 및 마감시간 초과 비율을 비교한다. 이 실험들을 통해 제안하는 *2PL-FT* 알고리즘이 기존에 제안된 방법들보다 더 우수함을 보인다.

Abstract

Real-time database systems (*RTDBS*) are database systems whose transactions are associated with timing constraints such as deadlines. Therefore transaction needs to be completed by a certain deadline. Besides meeting timing constraints, a *RTDBS* needs to observe data consistency constraints as well. That is to say, unlike a conventional database system, whose main objective is to provide fast average response time, *RTDBS* may be evaluated based on how often transactions miss their deadline, the average lateness or tardiness of late transactions, the cost incurred in transactions missing their deadlines. Therefore, in RTDBS, transactions should be scheduled according to their criticalness and tightness of their deadlines, even if this means sacrificing fairness and system throughput. And it always must guarantee preceding process of the transaction with the higher priority.

In this paper, we propose an efficient real-time scheduling algorithm (*Multi-level EFDF*) that alleviates problems of the existing real-time scheduling algorithms, a real-time concurrency control algorithm (*2PL-FT*) for firm and soft real-time transactions. And we compare the proposed *2PL FT* with *AVCC* in terms of the restarting ratio and the deadline missing ratio of transactions. We show through experiments that our algorithms achieve good performance over the other existing methods proposed earlier.

Keyword : Real-time database, Real-time Concurrency control, Feasibility Test

* 준 회 원 : 충북대학교 정보통신공학과 박사과정
sjlee@netdb.chungbuk.ac.kr(제 1저자)
** 준 회 원 : 충북대학교 정보통신공학과 석사과정
prettysam@netdb.chungbuk.ac.kr(공동저자)
*** 정 회 원 : 충북대학교 정보통신공학과 박사과정
segi21@netdb.chungbuk.ac.kr(공동저자)
**** 정 회 원 : 충북대학교 전기전자컴퓨터공학부 부교수
yjs@cbucc.chungbuk.ac.kr(공동저자)
☆ 본 연구는 한국과학재단 목적기초연구 (특정기초연구 과제번호 R01-2003-000-10627-0) 지원으로 수행되었음.

1. 서 론

트랜잭션이 마감시간(deadline)과 같은 시간제약조건을 가지는 시스템을 실시간 시스템(real-time system)이라 한다[1,2]. 대표적인 응용 예로는 응급의료, 항공, 군사 전략, 재난 방제, 공장 자동화, 로보틱스, 원자력 발전소, 교통 제어, 네트워크 서

비스 등이 있으며, 보다 다양한 분야에서 실시간 시스템의 활용도가 크게 증가되고 있는 추세이다[3~7]. 또한 많은 실시간 응용들이 비정형, 대용량, 시간성 등의 특성을 갖는 멀티미디어 데이터를 적극적으로 수용하고 원활한 서비스를 가능하게 하는 실시간 데이터베이스 시스템(real-time database system)으로 확장되고 있다.

실시간 데이터베이스의 트랜잭션은 마감시간 내에 처리가 완료되지 못하는 경우 '처리 결과의 값이 유효한지?' 그리고 '생명과 재산상의 손실 및 재앙을 초래하는지?' 여부에 따라 하드(hard), 펌(firm), 소프트(soft) 실시간 트랜잭션으로 분류된다[8~10].

먼저, 하드 실시간 트랜잭션이란 마감시간 내에 완료되지 못하는 경우에 시스템에서 즉시 제거되는 것뿐만 아니라 치명적인 손실 및 큰 재앙을 초래하는 것을 일컫는다. 다음으로, 펌 실시간 트랜잭션이란 마감시간을 지키지 못하는 경우에 작업 결과의 가치는 소멸되지만, 치명적인 손실을 초래하지는 않는 것을 일컫는다. 마지막으로, 소프트 실시간 트랜잭션이란 마감시간을 초과하더라도 결과의 가치가 다소 떨어질 뿐이며, 치명적인 손실도 초래하지 않는 것을 말한다[11~13].

일반적으로 하드 실시간 트랜잭션은 주기적으로 반복되는 작업으로 읽기/쓰기 집합이 미리 알려져 있는 것이 대부분이다. 반면에 펌 또는 소프트 실시간 트랜잭션은 비주기적인 작업들로 실행시간 및 데이터 요구 사항을 미리 예측하기 어렵다. 따라서 펌 또는 소프트 실시간 트랜잭션을 다루는 일반적인 실시간 데이터베이스 시스템에서는 제한된 시간 내에 보다 많은 수의 트랜잭션을 처리하여 전체 트랜잭션의 마감시간 초과 비율이 최소화되도록 지원하는 것이 필요하다[14~16].

최근까지, 트랜잭션의 마감시간 초과 비율을 최소화하기 위한 목적으로 다양한 실시간 병행실행제어 알고리즘들이 제안되었다. 그러나 기존의 알고리즘들은 트랜잭션의 마감시간 또는 여유시간만을 고려하고 있어 개선이 요구된다. 또한 낮은 우선순위 트랜잭션을 철회 또는 대기시키고 선행처리중인 높은 우선순위 트랜잭션이 마감시간을 지키지 못하고 결국 시스템에서 제거되는 경우도 발생된다. 이와 같은 상황에서는 낮은 우선순위를 갖는 트랜잭션의 철회 및 대기가 불필요한 일이 되어 버리므로 자원의 낭비를 초래한다.

위에서 언급한 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회 문제를 해결하기 위해 최근에 대체버전을 이용한 병행실행제어 방법이 제안되었다[22]. 이 방법에서는 충돌 발생 시 낮은 우선순위 트랜잭션에 대한 재시작 버전과 대기 버전을 동시에 유지한다. 그리고 상황에 따라 유리한 버전을 사용한다. 그러나 항상 두 개의 버전을 동시에 유지하면서 결국 하나의 버전만이 사용되기 때문에 자원의 낭비를 초래할 수밖에 없다. 또한 재시작 버전을 이용하여 새롭게 작업을 실행하는 도중에 또 다른 충돌이 발생되어 하나의 트랜잭션에 대한 제 2, 제 3의 재시작 버전이 생길 수가 있다. 그러므로 여러 재시작 버전들을 관리하기 위한 계층적 구조를 갖는 복잡한 버전 관리가 추가적으로 요구된다. 따라서 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회 및 불필요한 대기 문제를 보다 효율적으로 해결할 수 있는 새로운 병행실행제어 방법이 절실히 요구되고 있다.

본 논문에서는 기존 연구 결과에서 해결하지 못하고 있는 문제점들을 종합적으로 분석하고, 실시간 데이터베이스 시스템에 적합한 효율적인 병행실행제어 방법을 제시하고자 한다.

본 논문의 구성은 다음과 같다. 제 2장 관련 연구에서는 기존에 제안된 병행실행제어 알고리즘에 대해 살펴본다. 그리고 기존 방법들이 갖는 문제점들과 개선해야할 사항들을 자세하게 분석하고, 이에 대한 개선 방향을 제시한다. 제 3장에서는 먼저, 펌 실시간 트랜잭션이 마감시간 내에 정상적으로 완료될 수 있는지 여부를 판단하는 실행가능성 검사 방법과 이 검사에 필요한 예상 실행시간을 계산해 내는 방법에 대해 자세하게 기술한다. 그리고 예상실행시간 계산의 오차 문제

를 해결하기 위한 방안에 대해서도 추가적으로 기술한다. 둘째, 제안하는 실시간 병행실행제어 알고리즘에 대해 전체적인 구조 및 처리 방법에 대해 자세하게 소개한다. 4장에서는 제안하는 알고리즘들에 대한 성능을 기존의 방법들과 비교 분석한 결과를 제시한다. 마지막 5장, 결론 부분에서는 본 연구에 대한 종합적인 분석 결과와 연구 성과에 대해 기술하고, 향후 연구 방향을 제시한다.

2. 관련 연구

응급의료 시스템, 항공, 군사 전략 시스템, 재난 방제 시스템 등과 같은 많은 실시간 데이터베이스 시스템 응용에서 실시간 병행실행제어 알고리즘에 대한 연구는 매우 중요한 비중을 차지한다. 본 장에서는 실시간 데이터베이스 시스템을 위한 실시간 병행실행제어 알고리즘에 대해 기존 연구 내용을 조사 분석한다.

2.1 실시간 병행실행제어 알고리즘

실시간 병행실행제어 알고리즘에 대한 기존 연구는 충돌 검사와 재시작이 충돌 발생 시점에서 실행되는지, 아니면 완료 직전에 실행되는지에 따라 크게 비관적 병행실행제어(pessimistic concurrency control) 방법과 낙관적 병행실행제어(optimistic concurrency control) 방법으로 나눌 수 있다. 각각의 방법에 대해 자세하게 소개하면 다음과 같다.

(1) 비관적 병행실행제어

일반 데이터베이스 시스템에서 사용하는 2단계 잠금(two phase locking) 방식을 실시간 데이터베이스 응용에 이용하는 경우 높은 우선순위를 갖는 트랜잭션이 낮은 우선순위를 갖는 트랜잭션에 의해 대기하게 되는 우선 순위 역전(priority inversion) 문제가 발생될 수 있다. 게다가 트랜잭션의 우선순위를 전혀 고려하지 않기 때문에 실시간 데이터베이스 시스템을 위한 병행실행제어 방법으로는 적합하지 않다.

기존의 2단계 잠금 방식에 실시간 특성을 결합한 2PL-HP(two phase locking with high priority) 방법은 이미 자원을 점유하고 있는 트랜잭션과 해당 자원을 요청하는 트랜잭션간의 충돌 문제를 해결하기 위해 그림 1과 같이 동작한다. 처리 과정을 요약해 보면 다음과 같다.

어떤 자원 *O*에 대한 잠금을 소유한 트랜잭션(*TH*)과 자원 *O*의 잠금을 요청하는 트랜잭션(*TR*) 간에 충돌이 발생되는 경우, 먼저 두 트랜잭션의 우선순위를 비교한다. 만약 잠금을 요청하는 트랜잭션(*TR*)의 우선순위가 더 높은 경우, 잠금을 소유한 낮은 우선순위 트랜잭션(*TH*)을 철회시킨 후 높은 우선순위를 갖는 트랜잭션(*TR*)을 실행시킨다. 위와 반대로 자원을 소유한 *TH*의 우선순위가 더 높은 경우, *TH*는 계속해서 실행되고 *TR*은 대기하게 된다. 즉, 2PL-HP 방법은 충돌 발생 시 항상 낮은 우선순위를 갖는 트랜잭션(*LPT*)을 철회 또는 대기시키고, 높은 우선순위 트랜잭션(*HPT*)의 선행 처리를 항상 보장하기 때문에 우선순위에 따른 실행이 필수적인 실시간 응용에 적합한 방법이다.

그런데 2PL-HP 방법에서 최소 여유시간 우선 방식(least slack first)을 사용하여 우선순위를 할당하는 경우, 트랜잭션의 연속적인 철회 및 재시작

```
/*  TH : 어떤 자원 O에 대한 잠금을 소유한 트랜잭션
    TR : 자원 O의 잠금을 요청하는 트랜잭션
    p(TH) : 잠금을 소유한 트랜잭션의 우선순위
    p(TR) : 잠금을 요청하는 트랜잭션의 우선순위 */
if (p(TH) < p(TR)) {
      Abort  TH;      /* 낮은 우선순위 트랜잭션을 중지시킴 */
      Run    TR;      /* 높은 우선순위 트랜잭션을 실행시킴 */
} else {
      TR   blocks;    /* 낮은 우선순위 트랜잭션을 대기시킴 */
      Run  TH;        /* 높은 우선순위 트랜잭션을 실행시킴 */
}
```

(그림 1) 우선 순위에 기반한 2단계 잠금 방법 (2PL-HP)

```
/*  TH : 어떤 자원 O에 대한 잠금을 소유한 트랜잭션
    TR : 자원 O의 잠금을 요청하는 트랜잭션
    p(TH) : 잠금을 소유한 트랜잭션의 우선순위
    p(T'H) : 잠금을 소유한 트랜잭션이 재시작 되어 새롭
             게 부여받을 우선순위
    p(TR) : 잠금을 요청하는 트랜잭션의 우선순위  */
if (p(TH) < p(TR) AND p(T'H) < p(TR)) {
      Abort  TH;
      Run    TR;
} else {
      TR   blocks;
      Run  TH;
}
```

(그림 2) 2PL-HP에서 재시작 트랜잭션에 대한 우선순위

문제가 발생될 수 있다. 이와 같은 문제를 해결하기 위해서는 그림 2와 같은 충돌 해결 방법이 요구된다. 즉, 충돌 발생 시 *TR*의 우선순위는 *TH*보다 더 높아야 될 뿐만 아니라 *TH*가 재시작 되어 새롭게 부여받을 우선순위보다도 더 높아야 한다는 것이다. 이와 같은 조건이 만족되는 경우에만 *TH*를 철회시키고 *TR*이 자원에 대한 잠금을 소유하게 한다[14].

다음으로 조건부 재시작(conditional restart) 방법은 그림 3과 같다. 충돌 발생 시 자원에 대한 잠금을 소유하고 있는 *LPT*의 남아 있는 실행시간이 *HPT*의 여유시간 보다 적은 경우, 실행중인 *LPT*가 계속 실행되도록 *HPT*가 대기한다. 즉, 높은 우선순위를 갖는 트랜잭션이 무조건 낮은 우선순위 트랜잭션을 철회시키는 것이 아니라 충분한 여유시간이 있다고 판단되는 경우 *LPT*가 완료될 수 있도록 기다려주는 방법이다.

이 방법을 사용하기 위해서는 잠금을 소유하고 있는 *LPT*의 남은 실행시간과 자원을 요청하는 *HPT*의 여유시간을 정확하게 예측할 수 있어야 한다. 그런데 *HPT*를 대기시키고 실행 중이던 *LPT*가 제 3의 트랜잭션에 의해 자원을 빼앗기고 철회되는 경우, 대기중인 *HPT*는 실행이 불가능해지는 단점을 갖는다.

(2) 낙관적 병행실행제어

낙관적 병행실행제어 방법에서 트랜잭션의 처리

```
/* 잠금을 요청하는 트랜잭션(TR)의 우선순위가 TH보다 더 높
  고, TH가 재시작 후 새롭게 부여받을 우선순위보다 더 높은
  경우 */
if (p(TH) < p(TR) AND p(T'H) < p(TR)) {
    /* TH가 완료될 때까지 TR이 기다릴 수 있는 경우 */
    if (remaining time of TH ≤ slack time of TR) {
        TR blocks;
        TH inherits the priority of TR;
        Run TH;
    } else {                    /* TR의 여유시간이 충분하지
                                   않은 경우 */
        Abort TH;
        Run  TR;
    }
} else {
    TR  blocks;
    Run TH;
}
```

(그림 3) 조건부 재시작 방법

과정은 크게 읽기 단계(read phase), 검사 단계(validation phase) 그리고 쓰기 단계(write phase)의 3단계로 구성된다. 읽기 단계에서는 트랜잭션 실행에 필요한 데이터를 각 트랜잭션의 로컬 영역으로 복사한 후 처리한다. 검사 단계에서는 트랜잭션이 사용한 자원에 대한 충돌 검사를 실시한다. 마지막으로 쓰기 단계에서는 완료가 보장된 트랜잭션에 대해 로컬 영역의 데이터를 데이터베이스에 반영한다. 검사 단계에서의 충돌 검사 방법은 이미 완료된 트랜잭션들을 대상으로 충돌 여부를 판별하는 역방향 검사(backward validation)와 이와 반대로 현재 실행중인 트랜잭션들을 대상으로 충돌 여부를 판별하는 순방향 검사(forward validation)가 있다. 검사 단계에 있는 트랜잭션과 읽기 단계에 있는 하나 이상의 트랜잭션(들) 중에 어떤 트랜잭션들을 철회할 것인지를 결정하는 방법에는 OPT-BC, OPT-Sacrifice, OPT-Wait, Wait-50, Wait-X 등이 있다[8,15,17]. 충돌 해결 방법들에 대해 보다 자세하게 살펴보면 다음과 같다.

먼저 OPT-BC(broadcast commit) 방법은 검사 단계의 트랜잭션이 충돌되는 다른 트랜잭션들과 비교하여 가장 낮은 우선 순위를 갖지만 않는다면 완료를 보장하고, 나머지 충돌되는 트랜잭션들을

모두 재시작 시킨다. 따라서 읽기 단계에 있는 *HPTs*가 검사 단계에 먼저 진입한 *LPT*에 의해 재시작 될 수 있으므로 스케줄링 시 *HPT*가 검사 단계에 빠르게 도착할 수 있도록 고려해야 한다.

OPT-Sacrifice 방법은 검사 단계 트랜잭션과 충돌하는 상위 트랜잭션, *CHP*(conflict higher priority)가 하나 이상 존재하는 경우 검사 단계 트랜잭션을 재시작 시킨다. 즉, 검사 단계에 진입한 트랜잭션이 충돌하는 다른 트랜잭션들 보다 더 높은 우선순위를 갖는 경우에만 완료가 보장된다.

OPT-Wait 방법은 검사 단계의 *LPT*가 불필요하게 재시작 되는 것을 줄이기 위한 방법으로 제안되었다. 검사 단계의 *LPT*와 충돌하는 *CHP* 트랜잭션들이 읽기 연산만을 갖는 경우, *LPT*는 모든 *CHP* 트랜잭션들이 끝나기를 기다린 후 더 이상 *CHP* 트랜잭션이 존재하지 않을 때 완료를 보장받는다. 그러나 *LPT*가 대기하는 동안 또 다른 *CHP* 트랜잭션이 추가될 수 있으므로 결국 재시작 되거나 또는 마감시간을 초과하여 철회되는 상황이 발생되기도 한다.

Wait-50 방법은 *CHP* 트랜잭션이 50% 미만인 경우에 검사 단계 트랜잭션의 완료가 보장되고, 그렇지 않은 경우 재시작 된다. 이렇게 함으로써 중·상위 우선 순위를 갖는 트랜잭션들의 무조건적인 재시작을 줄일 수 있다. Wait-X 방법은 Wait-50 방법을 시스템 환경에 따라 30% 또는 70% 등으로 다양하게 변화시킬 수 있도록 확장한 방법이다.

지금까지 살펴본 낙관적 병행실행제어 방법들은 트랜잭션의 처리가 끝난 이후에 검사 단계에서 충돌 검사 및 해결이 이루어지므로 블로킹(blocking)이 발생되지 않는다는 장점을 갖는다. 그러나 충돌 문제를 항상 트랜잭션의 재시작으로만 해결하기 때문에 블로킹과 재시작을 적절히 이용하는 2PL-HP 방법에 비해 불필요한 재시작이 많다. 따라서 이 방법은 자원 경쟁이 심하지 않은 경우에 유리하고, 반대로 자원 경쟁이 심한 경우에는 2PL-HP와 같은 비관적 병행실행제어 방법이 더 유리하다.

```
/* 충돌 발생 시 HPT는 실행, LPT는 즉시 재시작, 정지/재개 버전을 동시에 유지. */
if (HPT conflict with LPT) {
      Run HPT;
      Run LPT's IR version;
      Waiting for Running LPT's DR version;
}
/* 만약 HPT가 마감시간 내에 정상 완료되는 경우, LPT의 즉시 재시작 버전을 이용.
   반대로 HPT가 마감시간 내에 완료 못하는 경우, LPT의 정지/재개 버전을 실행. */
if (HPT commit)
      Run LPT's IR version;
else
      Run LPT's DR version;
```

(그림 4) 대체버전을 이용한 병행실행제어 방법 (AVCC)

(3) 대체버전을 이용한 병행실행제어

2PL-HP 방법을 사용하는 경우 불필요한 철회와 불필요한 대기 문제가 시스템에 큰 영향을 미친다[18]. 그리고 낙관적인 병행실행제어 방법을 사용하는 경우에는 낭비적인 실행 문제가 발생된다. 여기에서 낭비적인 실행이란 *LPT*가 먼저 검사 단계에 도달한 경우라 할지라도 준비 단계에서 실행중인 *HPT*에 의해 재시작 되는 경우를 말한다. 2PL-HP 방법에서 발생되는 불필요한 철회 문제는 충돌 발생 시 자원을 소유한 *LPT*를 철회하고 실행한 *HPT*가 마감시간을 지키지 못하고 결국 시스템에서 제거되는 경우에 발생된다. 또한 불필요한 대기 문제는 마감시간을 지키지 못하는 *HPT*에 의해 자원을 요청하는 *LPT*가 불필요하게 대기하는 경우에 발생된다.

위에서 언급한 불필요한 철회 문제를 해결하기 위해 대체버전을 이용한 병행실행제어 방법[18]이 최근에 제안되었다. 이 방법에서는 낙관적 병행실행제어 방법의 보류된 갱신을 채택하여 자원을 요청하는 *HPT*가 자원을 점유하고 있는 *LPT*를 즉시 재시작 시키지 않고 실행을 일시적으로 보류시킨다. 즉, *LPT*의 실행을 잠시 정지시켰다가 *HPT*가 비정상적으로 종료하는 경우 *LPT*를 다시 재개시킨다.

그림 4와 같이 앞서 실행 중이던 *LPT*와 동일

자원에 대한 잠금을 요청하는 *HPT*가 충돌하게 되면, *LPT*를 철회시키지 않고 정지/재개(Deferred Restart) 버전으로 계속 유지시킨다. 이와 동시에 *LPT*에 대한 즉시 재시작(Immediately Restart) 버전을 생성하여 *HPT*가 정상적으로 완료되는 경우를 대비하게 한다. *LPT*의 정기/재개 버전은 *HPT*가 비정상적으로 완료되는 경우를 대비하기 위한 것으로 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회를 방지할 수 있게 한다. 지금까지 살펴본 바와 같이 대체버전을 이용한 병행실행제어 방법은 충돌이 발생되면 항상 *LPT*에 대한 두 가지 버전을 동시에 유지하면서 상황에 따라 적절한 버전을 이용함으로써 낮은 우선순위 트랜잭션의 불필요한 철회 문제를 해결하고 있다.

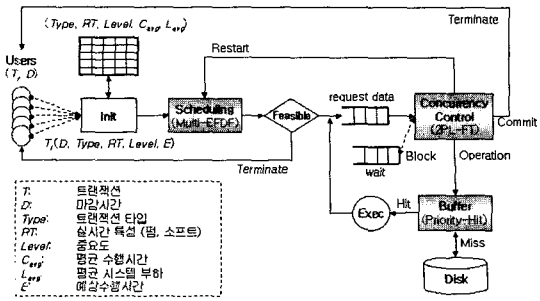
그러나 이 방법은 낮은 우선순위 트랜잭션의 불필요한 철회 및 실행 문제를 해결하기 위해 충돌 발생시마다 두 개의 버전(즉시 재시작 버전과 정지/재개 버전)을 동시에 유지해야 하는 단점을 갖는다. 또한 즉시 재시작 버전이 실행되는 동안 또 다른 *HPT*와 충돌이 발생될 수 있으므로 하나의 *LPT*에 대한 제 2, 제 3의 즉시 재시작 버전이 생성될 수 있다. 이와 같은 경우 즉시 재시작 버전들에 대한 복잡한 버전 관리가 추가적으로 필요하게 된다.

3. 실시간 스케줄링 및 병행실행제어 알고리즘

본 장에서는 제안하는 실시간 데이터베이스 시스템 모델의 주요 구성 요소들에 대해 자세하게 기술한다. 먼저 마감시간 내에 처리가 불가능한 트랜잭션을 선별해내기 위한 실행가능성 검사 방법을 제안한다. 그리고 실행가능성 검사 방법을 적용하여 불필요한 자원의 낭비를 방지할 수 있는 새로운 병행실행제어 알고리즘을 제안한다.

3.1 개요

제안하는 실시간 데이터베이스 시스템 모델의



(그림 5) 제안하는 실시간 데이터베이스 시스템 모델의 구성

주요 구성 요소는 그림 5와 같이 초기화, 실행가능성 검사, 그리고 병행실행제어 부분으로 나누어 볼 수 있다.

첫째, 초기화 부분은 마감시간을 부여받고 시스템에 진입한 새로운 트랜잭션에 대해 관련 정보를 얻는 단계이다. 즉, 트랜잭션 관리테이블을 참조하여 해당 트랜잭션의 타입 정보와 펌 또는 소프트와 같은 실시간 특성 정보 그리고 긴급성 및 중요도에 따른 레벨 정보를 획득한다. 또한 예상실행시간 계산에 필요한 평균 실행시간과 평균 시스템 부하 정보를 얻어온다. 그리고 이러한 정보를 바탕으로 해당 트랜잭션의 예상실행시간을 계산하게 된다. 트랜잭션이 정상적으로 완료되는 경우 실제 실행시간 및 실행 당시의 시스템 부하 정보는 트랜잭션 관리테이블에 추가된다.

둘째, 실행가능성 검사란 실행중인 트랜잭션이 마감시간 내에 완료될 수 있는지를 평가하는 것이다. 초기화 부분에서 각 트랜잭션은 평균 실행시간과 시스템 부하 정보를 바탕으로 예상실행시간을 부여받게 되므로 이 정보와 마감시간을 비교하여 트랜잭션이 정해진 시간 내에 완료가 가능한지를 평가할 수 있다. 실행가능성 검사는 불필요한 시스템 자원의 낭비를 방지하기 위한 것으로 트랜잭션 실행 시작 시점과 충돌 발생 시점에서 실행된다. 트랜잭션 실행 시작 시점(재 시작되는 트랜잭션 포함)에서는 모든 펌 실시간 트랜잭션을 대상으로(소프트 실시간 트랜잭션은 마감시간을 지키지 못하는 경우에도 결과가 인정되므

로 검사 대상에서 제외) 실행가능성 검사를 실시한다. 이때 마감시간 내에 완료가 불가능한 트랜잭션들을 모두 제거시킴으로써 불필요한 자원의 낭비를 막는다. 그리고 충돌 발생 시점에서는 두 트랜잭션 중에서 높은 우선순위를 갖는 펌 실시간 트랜잭션에 대해서만 실행가능성 검사를 실시한다. 이를 통해 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회 및 대기 문제를 해결하게 된다. 만약 충돌 발생 시 소프트 실시간 트랜잭션이 높은 우선순위를 갖는 경우에는 앞서 실행중인 트랜잭션의 계속적인 실행을 보장하고 자원을 요청하는 (소프트 실시간) 트랜잭션은 선행 트랜잭션의 처리가 끝날 때까지 대기한다.

셋째, 실시간 병행실행제어 알고리즘은 높은 우선순위를 갖는 트랜잭션의 선행 처리를 항상 보장하되, 불필요한 실행은 막는다. 따라서 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회 및 대기 문제를 해결한다. 예를 들어 높은 우선순위를 갖는 펌 실시간 트랜잭션이 자원을 소유하고 있는 상태에서 해당 자원을 낮은 우선순위 트랜잭션이 요청하는 경우, 먼저 높은 우선순위를 갖는 트랜잭션의 실행가능성 검사가 이루어진다. 만약 검사 결과가 참이면 자원을 요청한 낮은 우선순위 트랜잭션을 대기하게 한다. 그러나 거짓인 경우에는 자원을 소유한 높은 우선순위 트랜잭션이 마감시간 내에 완료될 수 없는 상황이므로 이 트랜잭션을 시스템에서 제거시키고 자원을 요청하는 낮은 우선순위 트랜잭션에게 자원을 할당한다. 이를 통해 낮은 우선순위 트랜잭션의 불필요한 대기 문제가 해결된다.

앞의 예와 반대의 경우를 살펴보면 다음과 같다. 자원을 요청하는 트랜잭션이 높은 우선순위를 갖는 펌 실시간 트랜잭션인 경우, 이 트랜잭션에 대해 실행가능성 검사를 실시한다. 그 결과가 참이면 앞서 자원을 소유한 낮은 우선순위 트랜잭션을 철회시키고, 자원을 요청한 높은 우선순위 트랜잭션에게 자원을 할당한다. 반면에 거짓인 경우에는 높은 우선순위 트랜잭션에게 자원을 할당해도 마감시간 내에 처리가 불가능하므로 미리 시스템에서 제거시킨다. 이를 통해 앞서 실행 중이던 낮은 우선순위 트랜잭션의 불필요한 철회 문제를 해결할 수 있다. 게다가 높은 우선순위 트랜잭션의 불필요한 실행을 막을 수 있으므로 전체적인 마감시간 초과 비율을 크게 줄일 수 있는 장점을 갖는다.

위에서 언급한 바와 같이 제안하는 실시간 데이터베이스 시스템 모델에서는 각 트랜잭션별로 자신의 타입 정보와 펌 또는 소프트와 같은 실시간 특성 정보 그리고 중요도에 따른 레벨 정보를 갖는다고 하였다. 이를 위해서는 특정 응용에서 발생 가능한 트랜잭션의 종류는 무한하지 않고, 일정 개수의 트랜잭션 타입으로 분류가 가능하다고 가정한다.

3.2 실행가능성 검사 방법

실행가능성 검사는 실행을 시작하는 펌 실시간 트랜잭션이 마감시간 내에 처리될 수 있는지 여부를 미리 검사하여 불필요한 실행을 방지하는 역할을 실행한다. 또한 트랜잭션 실행 도중에 자원을 소유하고 있는 트랜잭션과 자원을 요청하는 트랜잭션 사이에 충돌이 발생되는 경우에도 높은 우선순위를 갖는 펌 실시간 트랜잭션을 대상으로 실행가능성 검사를 실행한다. 이를 통해 마감시간 내에 처리가 완료될 수 없는 트랜잭션이 불필요하게 자원을 낭비하는 것을 막고, 낮은 우선순위를 갖는 트랜잭션이 불필요하게 철회 또는 대기하는 것을 막는다.

(1) 실행가능성 검사 시점에 따른 분류

제안하는 방법에서 실행가능성 검사는 트랜잭션 실행 시작 시점과 충돌 발생 시점 모두에서 실행된다. 만약, 트랜잭션 실행 시작 시점에서만 검사를 실행하는 방법을 사용한다면 여러 번 충돌이 발생되어 계속해서 대기 시간이 증가하는 경우 트랜잭션 실행 시작 시점의 검사 결과는 가

치가 떨어지거나 또는 무의미하게 된다. 이와 같은 잘못된 판단 기준으로 트랜잭션의 계속 실행 및 철회 여부를 결정하는 것은 시스템에 큰 악영향을 끼치게 되므로 이에 대한 개선이 요구된다. 위와 달리 충돌 시점에서만 검사를 실행하는 방법은 위에서 언급한 문제점을 해결할 수 있다. 그러나 실행 시작 시점에서 마감시간 내에 완료가 불가능한 트랜잭션을 미리 제거할 수 없다는 단점을 갖는다.

지금까지 살펴본 바와 같이 실행가능성 검사 시점 측면에서 한 가지 방법만을 적용하는 경우, 여러 가지 문제점이 발생되는 것을 알 수 있다. 이에 본 논문에서는 두 가지 방법을 상호보완적으로 결합하여 시작 시점 및 충돌 시점 모두에서 실행가능성 검사를 실시한다.

(2) 처리 절차

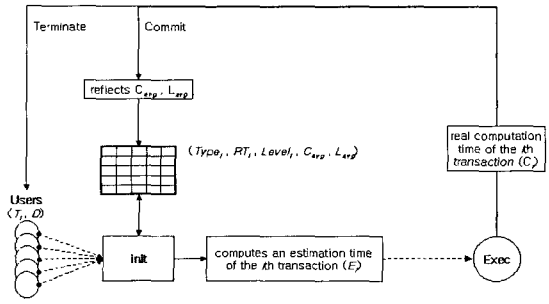
실행가능성 검사를 위해 필요한 파라미터들을 정리하면 표 1과 같다.

그리고 전체적인 실행가능성 검사 처리 절차는 그림 6과 같다.

각 트랜잭션(T_i)들은 마감시간(D_i)을 부여받고 시스템에 진입한다. 그리고 가장 먼저 트랜잭션 타입($Type_i$)에 해당하는 정보를 테이블 검색을 통해 획득하게 된다. 이를 통해 펌 또는 소프트와 같은 실시간 특성과 긴급성 및 중요도에 따른 레벨($Level_i$) 정보를 획득한다. 그리고 자신의 예상실

(표 1) 실행가능성 검사의 파라미터

파라미터	의 미
T_i	i번째 트랜잭션
D_i	i번째 트랜잭션의 마감시간
$Type_i$	i번째 트랜잭션의 타입
RT_i	i번째 트랜잭션의 실시간 특성(펌, 소프트)
$Level_i$	i번째 트랜잭션의 중요도
C_{avg}	해당 타입의 평균 실행시간
L_{avg}	해당 타입의 평균 시스템 부하
L_i	현재 시스템 부하
E_i	i번째 트랜잭션의 예상실행시간
C_i	i번째 트랜잭션의 실제 실행시간



(그림 6) 실행가능성 검사 처리과정

행시간 계산에 필요한 통계 정보인 해당 트랜잭션의 평균 실행시간(C_{avg})과 평균 시스템 부하(L_{avg})를 얻게 된다. 이 정보를 바탕으로 자신의 예상실행시간(E_i)을 구하게 된다. E_i 계산 방법은 $Level_i$ 값에 따라 2가지로 구분되어 처리된다. 마지막으로, 마감시간 내에 실행이 정상적으로 완료된 트랜잭션에 대해서만 실제 실행시간(C_{i-1})과 이 때의 시스템 부하(L_{i-1}) 정보를 각각 (수식 1), (수식 2)에 반영하여 다음 번 트랜잭션 실행에 필요한 평균 실행시간(C_{avg}) 및 평균 시스템 부하(L_{avg}) 정보를 계산해낸 후 그 결과를 테이블에 기록한다.

$$C_{avg} = \frac{C_{i-1} + C_{avg}}{2} \qquad \text{(수식 1)}$$

$$L_{avg} = \frac{L_{i-1} + L_{avg}}{2} \qquad \text{(수식 2)}$$

위와 같은 계산식을 이용하는 것은 계산이 간단하다는 장점과 더불어 시스템의 최근 상황을 반영하기 위해서이다.

그림 7은 실행가능성 검사 함수를 정의한 것이다. 마감시간(D_i)과 예상 실행시간(E_i)을 갖는 트랜잭션(T_i)이 t시점까지의 실제 실행시간(C_i)을 가지고 실행가능성 검사를 받게 되는 경우 (수식 3)과 같은 조건식에 의해 평가된다.

$$(D_i - t) - (E_i - C_i) < 0 \qquad \text{(수식 3)}$$

위 식과 같이 충돌 시점 t에서 마감시간까지의

```
/* 설명: 실행가능성 검사 함수
입력 값: 트랜잭션(T)
반환 값: 입력된 트랜잭션(T)이 마감시간 내에 완료가 가능한
경우 '1'을 반환하고, 그렇지 않은 경우 '0'을 반환함. */

int Feasibility_test(T) {
    clock_t t = clock();          /* 현재 시간          */
    if((T.deadline - t) - (T.estm_c - T.real_c) < 0)
            return 0;             /* T is infeasible    */
    else
            return 1;             /* T is feasible      */
}
```

(그림 7) 트랜잭션에 대한 실행가능성 검사 함수

시간 간격($D_i - t$)보다 앞으로 처리되어야 할 실행시간($E_i - C_i$)이 작은 경우, 정상적인 실행이 가능하므로 *True* 값을 반환한다. 그러나 마감시간 내에 처리가 불가능한 경우 *False*가 반환되어 트랜잭션 T_i는 시스템에서 제거되고 사용 중이던 자원은 모두 해제된다.

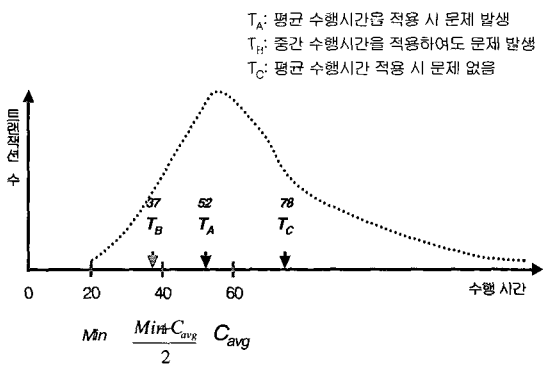
(3) 예상실행시간 계산 방법

접근해야 할 데이터 목록이 이미 정의되어 있는 하드 실시간 트랜잭션의 경우에는 데이터 처리를 위해 소요되는 시간, 입출력 시간, 데이터 잠금을 획득하고 해제하는 데 필요한 시간 및 발생 횟수, 스케줄링에 소요되는 시간 그리고 대기 시간 등을 미리 알 수 있으므로 해당 트랜잭션의 실제 실행시간을 거의 정확하게 계산할 수 있다[23,24].

그러나 제안하는 실시간 데이터베이스 시스템 모델처럼 접근해야 할 데이터 목록이 정해져 있지 않은 일반적인 소프트, 펌 실시간 트랜잭션을 다루는 환경에서는 트랜잭션의 예상실행시간을 예측하는 새로운 계산 방법이 요구된다. 그런데 예상실행시간은 미래에 대한 예측이기 때문에 아무리 실제 실행시간에 근접한 계산 결과를 도출해 낸다 하더라도 100% 정확한 예측은 불가능하다고 할 수 있다. 따라서 전체 트랜잭션 중에서 일부는 잘못된 예상실행시간을 할당받아 피해를 보게 되는 경우가 발생되기 마련이다. 본 논문에서는 이에 대한 예제를 분석하여 그 해결책을 제시한다.

그림 8은 어느 하나의 트랜잭션을 대상으로 실행시간 분포 그래프를 작성해 본 예이다. 대부분의 실행시간은 전체 평균값(C_{avg})을 중심으로 분포되어 있고, 시스템 부하 변화에 따라 평균보다 작거나 또는 큰 실행시간을 갖는 것을 알 수 있다. 트랜잭션의 실행시간은 시스템 부하가 전혀 없는 최적의 상황에서 실행된다 하더라도 최소실행시간(*Min*) 보다 더 빨리 처리될 수는 없다. 이와 반대로 시스템 부하가 급격하게 증가하는 경우 이에 비례해서 매우 큰 실행시간을 갖는 경우가 발생되기도 한다. 이 트랜잭션에게 부여된 마감시간에 따라 완료가 불가능한 트랜잭션은 시스템에서 제거되므로 무한대 시간까지 실행시간이 증가되는 경우는 발생되지 않는다.

그림 8에서 C_{avg}는 전체 실행시간 중에서 마감시간 내에 처리가 완료된 트랜잭션들만을 대상으로 누적 평균을 구한 값이다. 만약 이 값(C_{avg})을 새롭게 실행될 트랜잭션의 예상실행시간으로 설정하는 경우, 트랜잭션 T_A, T_B는 피해를 볼 수 있다. 왜냐하면 실행 도중에 충돌이 발생되어 트랜잭션 T_A 또는 T_B가 실행가능성 검사를 받게 되는 경우, 시스템은 C_{avg} 시간이 되어야 처리가 완료될 것이라고 판단할 것이다. 그런데 트랜잭션 T_A 또는 T_B는 실제로 C_{avg} 시간보다 더 빠르게 실행을 완료할 수 있는 상황이다. 그러나 시스템은 C_{avg}와 해당 트랜잭션의 마감시간을 비교하여 마감시간 내에 완료가 불가능하다고 판단되는 경우 트랜잭션



(그림 8) 실행시간에 따른 트랜잭션의 분포 예

T_A 또는 T_B를 시스템에서 제거시키게 된다.

위와 반대의 경우로 T_C는 C_{avg} 시간보다 더 긴 시간이 요구되는 트랜잭션이다. 충돌이 발생되어 실행가능성 검사를 실시하는 경우 시스템은 이 트랜잭션이 C_{avg} 시간이면 실행이 완료될 수 있을 것으로 판단한다. 따라서 트랜잭션 T_C는 비록 마감시간 내에 완료가 불가능한 경우라 할지라도 시스템에서 제거되지 않고 계속 실행된다.

위에서 살펴본 바와 같이 새롭게 실행될 트랜잭션에게 적합한 예상실행시간을 할당하는 문제는 매우 중요하다. 그림 8에서 볼 수 있듯이 단순하게 C_{avg}를 예상실행시간으로 설정하게 되면 트랜잭션 T_A, T_B와 같이 제거되지 말아야 할 트랜잭션이 불필요하게 제거되는 폐해가 발생된다. 따라서 예상실행시간을 C_{avg}보다 더 작은 값으로 결정할 필요가 있다.

만약, 해당 트랜잭션의 최소실행시간(*Min*)으로 예상실행시간을 설정하는 경우에는 트랜잭션 T_A, T_B처럼 불필요하게 제거되는 문제를 방지할 수는 있지만, 실행가능성 검사를 사용함으로써 얻어지는 효과가 크지 않게 된다. 이에, 본 논문에서는 $\frac{Min + C_{avg}}{2}$를 예상실행시간으로 사용한다. 이렇게 함으로써 트랜잭션 T_A와 같은 경우까지 고려될 수 있도록 하였다. 그러나 발생 빈도수가 많지 않은 T_B와 같은 트랜잭션이 불필요하게 제거되는 문제는 여전히 남아있다.

실시간 데이터베이스 시스템에서 가장 중요한 척도로 삼고 있는 것은 마감시간 초과 비율이다. 즉, 한정된 시스템 자원 하에서 보다 많은 실시간 트랜잭션을 마감시간 내에 처리하는 것이 중요하다는 것이다. 따라서 극소수의 트랜잭션이 피해를 볼 수는 있으나 전체적인 마감시간 초과 비율을 높이는 것은 의미 있는 일이다. 그런데 추가적으로 고려해 보아야 할 중요한 문제가 있다. 그것은 본 논문에서 강조하고 있는 것처럼 트랜잭션의 마감시간보다 더 우선적으로 고려해야 할 것은 트랜잭션의 중요도이다.

```
if(T.level != max_level)   /* 응용에 따라 변경 가능      */
        Type 1;            /* 일반 트랜잭션에 대한 처리 */
else
        Type 2;            /* 중요 트랜잭션에 대한 처리 */
```

(그림 9) 중요도에 따른 예상실행시간 계산 방법

극소수의 트랜잭션이 피해를 보더라도 더 많은 수의 트랜잭션이 마감시간을 지키게 하는 데에는 약간의 문제가 있다. 왜냐하면 피해를 당하는 극소수의 트랜잭션 중에 매우 중요한 트랜잭션이 존재할 수 있기 때문이다. 이에 대해 본 논문에서는 다음과 같은 해결 방법을 제시한다.

긴급하거나 중요도가 매우 높은 트랜잭션들은 별도로 관리되어야 한다. 이를 위해 제안하는 방법에서는 그림 9과 같이 해당 트랜잭션의 레벨 정보에 따라 가장 중요도가 높은 레벨의 트랜잭션인 경우 *Type 2*에 따라 예상실행시간을 계산하고, 그렇지 않은 경우 *Type 1*에 따라 예상실행시간을 계산하도록 하고 있다.

여기에서 *Type 1*은 레벨 정보에 따라 구분되는 중요도가 높지 않은 일반 트랜잭션을 대상으로 한 것으로 $\frac{Min + C_{avg}}{2}$에 시스템 부하 변동 비율을 반영하여 예상실행시간을 계산하는 방법이다(수식 4).

*Type 2*는 중요도가 가장 높은 트랜잭션을 대상으로 한 것으로 *Min* 값에 시스템 부하 변동율을 반영하는 방법이다(수식 5). 즉, 불필요한 제거 때문에 피해를 볼 수 있는 극히 적은 수의 트랜잭션 중에서 중요도가 높은 트랜잭션은 *Type 2*를 적용하여 정상적인 실행을 보장하고자 하는 것이다. 이로 인해 *Type 2*로 분류되는 중요한 트랜잭션의 경우 불합리하게 검사에서 피해를 보는 일은 없게 된다. 반면에 *Type 2*에 속한 트랜잭션들은 마감시간 내에 완료되지 못하는 경우에도 미리 제거되지 않으므로 불필요한 자원 낭비를 초래할 수 있다. 따라서 그림 9와 같이 중요도가 가장 높은 트랜잭션들에 대해서만 *Type 2*를 적용하는 것이 바람직하다.

Type 1: 일반 트랜잭션을 위한 예상실행시간 계산법

$$E_i = \frac{Min + C_{avg}}{2} \times \frac{L_i}{L_{avg}} \quad \text{(수식 4)}$$

Type 2: 중요 트랜잭션을 위한 예상실행시간 계산법

$$E_i = \max(Min, Min \times \frac{L_i}{L_{avg}}) \quad \text{(수식 5)}$$

(수식 4)와 (수식 5)에서 $\frac{L_i}{L_{avg}}$는 시스템 부하 평균(L_{avg})에 비해 트랜잭션 T_i가 실행될 당시의 시스템 부하(L_i)가 더 큰지, 아니면 작은지를 나타내는 것으로 시스템 부하 변동 상황을 고려하여 현재 상황에 가장 적합한 예상실행시간을 계산해 내기 위해 사용된 것이다. 이를 통해 시스템 부하가 급격하게 증가 또는 감소하는 경우에도 예상실행시간 계산의 정확도를 높일 수 있다.

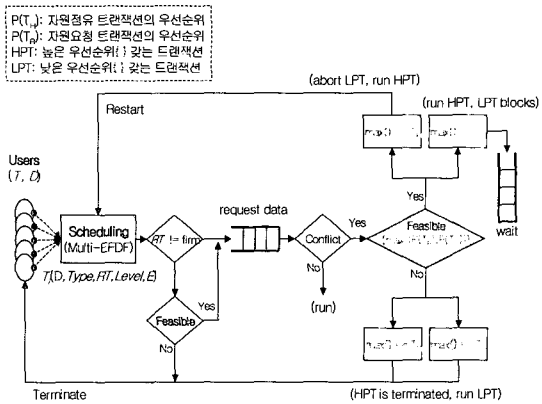
3.3 실행가능성 검사를 이용한 병행실행제어 알고리즘 (2PL-FT)

기존 연구의 문제점을 해결하기 위해 본 논문에서는 실행가능성 검사를 사용하는 새로운 실시간 병행실행제어 알고리즘을 제안한다.

(1) 충돌 해결 방법

펌 또는 소프트 실시간 트랜잭션을 대상으로 충돌 문제를 해결하는 새로운 실시간 병행실행제어 알고리즘인 2PL-FT(two phase locking with feasibility test)는 그림 10과 같은 구조를 갖는다.

가. Multi-level EFDF 스케줄링에 따라 가장 높은 우선순위를 갖는 트랜잭션부터 우선적으로 실행을 시작하게 된다. 이때 마감시간 내에 완료되지 못하는 펌 실시간 트랜잭션은 실행가능성 검사를 통해 미리 제거된다.



(그림 10) 제안하는 2PL-FT 알고리즘의 동작과정

나. 충돌 발생 시 자원을 점유하고 있는 트랜잭션(T_H)과 이 자원을 요청하는 트랜잭션(T_R) 중에서 높은 우선순위를 갖는 펌 실시간 트랜잭션을 대상으로 실행가능성 검사를 실시한다.

다. 자원을 점유하고 있는 높은 우선순위 트랜잭션(T_H)이 마감시간 내에 완료될 수 없다고 판정된 경우, 즉시 이 트랜잭션을 시스템에서 제거시킨다. 이로 인해 자원을 요청하는 낮은 우선순위 트랜잭션(T_R)은 불필요한 대기를 피할 수 있게 된다.

라. 위와 반대로, 자원을 요청하는 높은 우선순위 트랜잭션(T_R)이 마감시간 내에 완료될 수 없다고 판정된 경우, 이 트랜잭션은 시스템에서 제거된다. 이로 인해 자원을 점유한 낮은 우선순위 트랜잭션(T_H)은 불필요하게 철회되지 않고 계속 실행이 가능하다.

위에서 설명한 바와 같이 제안하는 2PL-FT 알고리즘은 높은 우선순위를 갖는 펌 실시간 트랜잭션의 실행가능성 검사를 통해 불필요하게 자원을 점유하고 있는 펌 실시간 트랜잭션을 제거시킬 수 있다. 이로 인해 낮은 우선순위를 갖는 트랜잭션의 불필요한 지연 문제가 해결되고 전체적

인 마감시간 초과 비율을 감소시킨다. 또한 제안하는 알고리즘은 불필요하게 자원을 요청하는 높은 우선순위를 갖는 펌 실시간 트랜잭션을 제거시킴으로써 실행 중이던 낮은 우선순위 트랜잭션의 불필요한 철회 문제를 해결하는 효율적인 방법이다.

추가적으로 고려해 보아야 할 문제는 소프트 실시간 트랜잭션들 상호간에 충돌이 발생되는 상황이다. 왜냐하면 소프트 실시간 트랜잭션은 마감시간 이후에 완료된다 하더라도 결과의 가치가 다소 떨어질 뿐 유효하기 때문에 앞서 실행 중이던 소프트 실시간 트랜잭션을 제거시키고 새로운 소프트 실시간 트랜잭션을 실행시키는 것은 크게 의미를 갖지 못한다. 오히려 앞서 실행중인 트랜잭션부터 순차적으로 실행시키는 것이 더 효율적이다. 따라서 제안하는 알고리즘에서는 충돌 발생시 높은 우선순위를 갖는 트랜잭션이 소프트 실시간 트랜잭션인 경우에는 실행가능성 검사를 생략하고 앞서 실행중인(소프트 또는 비 실시간)트랜잭션의 계속적인 실행을 보장한다. 알고리즘의 전체적인 동작 과정은 그림 11과 같다.

4. 성능 평가

성능평가는 대체버전을 이용한 병행실행제어 알고리즘을 비교 대상으로 하였다. 성능평가 척도로 사용한 것은 마감시간 초과 비율과 트랜잭션 재시작 비율이다. 마감시간 초과 비율이란 (수식 6)과 같이 전체 실행시킨 트랜잭션 중에서 마감시간 내에 완료되지 못하고 철회된 트랜잭션의 개수를 비율로 나타낸 것이다. 그리고 트랜잭션 재시작 비율은 전체 실행시킨 트랜잭션 중에서 충돌에 의해 재시작 된 트랜잭션의 개수를 비율로 나타낸 것이다.

$$\text{마감시간초과비율}(\%) = \frac{\text{철회된트랜잭션의수}}{\text{전체트랜잭션의개수}} \times 100$$

(수식 6)

```
/* 설명 두 트랜잭션의 충돌 문제를 해결하기 위한 함수.
   입력 값:  T1: 어떤 데이터 객체 O의 잠금을 소유한 트랜잭션
             T2: 데이터 객체 O의 잠금을 요청하는 트랜잭션
   반환 값:  없음 */

void Solve_conflict(T1, T2) {
 /* Compare_priority()는 두 트랜잭션의 우선순위를 비교하는 함
    수로 첫 번째 입력 파라미터인 T1이 더 높은 우선순위를 갖
    는 경우 '1'을 반환하고 그렇지 않은 경우 '0'을 반환함.
    Feasibility_test()는 트랜잭션의 실행가능성 검사를 실행하는
    함수로 입력 파라미터로 주어진 트랜잭션이 마감시간 내에
    완료될 수 있는 경우 '1'을 반환하고 그렇지 않은 경우 '0'을
    반환함. */
if (Compare_priority(T1, T2)) { /* 첫 번째 파라미터인 T1이 더 높은
   우선순위를 갖는 경우*/
    if (Feasibility_test(T1)) {  /* if T1 is feasible */
        T1 continues;
        T2 blocks;
    } else {                     /* if T1 is infeasible */
            Abort T1;
            Run T2;
    }
} else { /* 두 번째 파라미터인 T2가 더 높은 우선순위를 갖는
          경우 */
  if (Feasibility_test(T2)) {          /* if T2 is feasible */
            Abort T1;
            Run T2;
  } else {                             /* if T2 is infeasible */
            Abort T2;
            T1 continues;
           }
  }
}
```

(그림 11) 두 트랜잭션의 충돌 문제를 해결하기 위한 함수

표 2는 성능평가에 사용된 시스템 파라미터를 보여준다.

성능평가에 사용된 컴퓨터는 인텔 펜티엄 III 프로세서, 1GHz, 256MB이며, OS는 윈도우즈 2000, 컴파일러는 비주얼 C++ 6을 사용하였다. 성능 평가에 사용된 시스템 파라미터는 표 2와 같다. 이것은 비교 대상인 대체버전을 이용한 병행실행제어 알고리즘에서 사용한 성능평가 환경을 이용한 것이다. 시뮬레이션에 사용한 트랜잭션은 마감시간, 중요도 등과 같은 속성 값을 갖는 펌 실시간 트랜잭션들로 구성하였다. 성능평가를 위해 100가지

(표 2) 시스템 파라미터

파라미터	내 용	설정 값
DBSize	데이터베이스 크기	1000 (pages)
CPUtime	페이지 처리 시간	10 (ms)
UpdateRatio	쓰기 비율	25 (%)
NumLevels	우선순위 레벨 수	20 (개)
NumTrans	트랜잭션 수	20000 (개)
NumTypes	트랜잭션 타입 수	100 (개)
TransSize	트랜잭션당 접근 페이지 수	8~24 (pages)
Slack	트랜잭션의 여유시간	100~600 (%)
Deadline	트랜잭션의 마감시간	도착시간+실행시간*Slack

타입의 트랜잭션을 정의하였고, 총 20,000개의 트랜잭션을 실행시켜 보았다. 그리고 초당 트랜잭션 도착율을 10부터 1,000까지 증가시켜 가면서 성능 평가를 실행하였다. 트랜잭션 실행 중에 접근하는 데이터 페이지 수는 8~24개로 균등 분포를 갖게 하였고, 여유시간은 실행시간의 100~600%로 하였다. 시뮬레이션 절차는 다음과 같다.

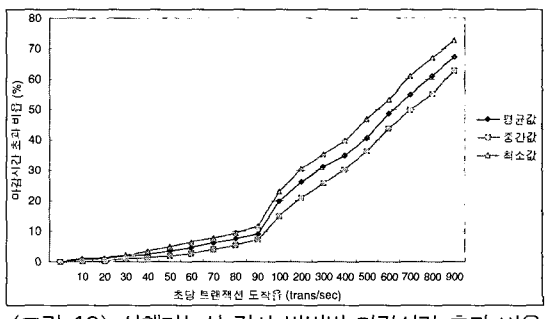
새로운 트랜잭션은 해당 타입별로 중요도가 결정되고 여유시간을 고려하여 마감시간이 부여된다. 그리고 각 트랜잭션 타입별로 중요도 레벨, 평균 실행시간, 최소 실행시간 그리고 시스템 부하 변동비율을 고려하여 예상실행시간이 산출된다. 스케줄러는 트랜잭션의 중요도에 따라 우선순위 레벨 큐를 결정하고 상위 레벨 큐에서 마감시간이 빠른 순으로 실행시킨다. 새롭게 실행되는 트랜잭션 그리고 재시작하는 트랜잭션은 시작 시점에서 실행가능성 검사를 받게 된다. 이를 통해 마감시간 내에 완료되지 못하는 트랜잭션을 미리 시스템에서 제거시킨다. 트랜잭션 실행 도중에 충돌이 발생되면 충돌이 발생한 두 트랜잭션 중에서 높은 우선순위를 갖는 트랜잭션의 실행가능성 검사를 실시한다. 검사 결과, 마감시간 내에 완료가 불가능한 경우 다음과 같이 두 가지 경우로 나누어 처리한다.

먼저, 이 트랜잭션이 잠금을 소유하고 있는 트랜잭션인 경우에는 이 트랜잭션을 시스템에서 제거시키고 잠금을 요청하는 낮은 우선순위를 갖는 트랜잭션이 즉시 실행될 수 있도록 한다. 위와 반대로 잠금을 요청하는 높은 우선순위 트랜잭션이 마감시간 내에 완료될 수 없는 것으로 판정된 경우, 이 트랜잭션을 시스템에서 제거시키고 앞서 실행 중이던 낮은 우선순위를 갖는 트랜잭션이 계속 실행될 수 있도록 한다. 그리고 트랜잭션의 실행이 정상적으로 완료되면 이 때의 실행시간 정보를 기존의 평균 실행시간과 산술 평균하여 새로운 평균 실행시간으로 기록한다. 이 정보는 다음에 실행될 트랜잭션의 예상실행시간 계산에 이용된다.

실험은 3가지 측면에서 실시하였다. 먼저, 제안하는 2PL-FT 알고리즘에서 실행가능성 검사에 이용되는 예상실행시간 계산 방법들 간의 성능 비교를 실행하였다. 둘째, AVCC와 2PL-FT 알고리즘의 트랜잭션 재시작 비율을 비교해 보았다. 마지막으로, AVCC와 2PL-FT 알고리즘의 마감시간 초과 비율을 비교하여 제안하는 알고리즘의 성능 향상 정도를 알아보았다. 각 실험 결과에 대한 자세한 내용은 다음과 같다.

(1) 예상실행시간 계산 방법 비교

첫 번째 실험에서는 예상실행시간 계산 방법에 따른 마감시간 초과 비율을 비교한다. 실험 대상은 평균 실행시간을 이용하는 경우, 최소 실행시간을 이용하는 경우 그리고 이 두 값의 중간값을 이용하는 경우이다. 그림 13과 같이 중간값을 이용하는 경우에 가장 좋은 성능을 보이고 있음을

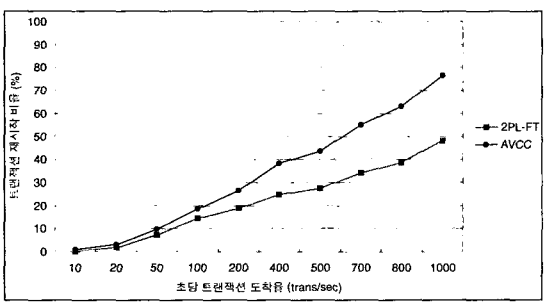


(그림 13) 실행가능성 검사 방법별 마감시간 초과 비율

알 수 있다. 반면에 최소값을 이용하는 경우가 가장 마감시간 초과 비율이 큰 것을 볼 수 있다. 이것은 세 가지 방법 중에서 최소값을 이용하는 경우에 예상실행시간 계산 오차가 가장 크기 때문이며, 이 오차로 인하여 실행가능성 검사에서 불필요하게 실행되는 뭄 실시간 트랜잭션을 제대로 선별해내지 못한 결과이다.

(2) 트랜잭션 재시작 비율 비교

두 번째 실험에서는 비교 대상인 AVCC와 제안하는 2PL-FT 알고리즘의 트랜잭션 재시작 비율을 비교한다. 그림 14와 같이 제안하는 알고리즘의 재시작 비율이 37.5% 정도 더 낮은 것을 알 수 있다. 왜냐하면 AVCC 방법은 낮은 우선순위를 갖는 트랜잭션이 자원을 점유하고 있는 상황에서 더 높은 우선순위를 갖는 트랜잭션이 해당 자원을 요청할 때, 항상 대기 버전과 1차 재시작 버전을 동시에 유지한다. 그리고 1차 재시작 버전이 실행되면서 또 다른 충돌로 인하여 대기 버전이 되고, 제 2차 재시작 버전을 생성하는 경우도 발생된다. 이렇게 생성되는 재시작 버전들은 자원 경쟁이 심할수록 더욱더 증가하게 되므로 복잡한 버전 관리 문제가 대두되기도 한다. 이와 같은 상황에서 자원을 빼앗아간 높은 우선순위 트랜잭션이 마감시간 내에 완료되지 못하고 시스템에서 제거되는 경우, 최초의 대기 버전이 자원을 획득하고 실행을 계속하게 된다. 이때 1, 2차 재시작 버전들은 더 이상 필요하지 않으므로 시스템에서 제거되어 버린다.

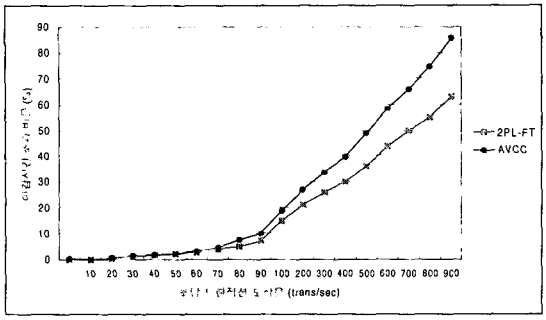


(그림 14) 트랜잭션 도착율에 따른 재시작 비율

이와 달리 제안하는 2PL-FT 알고리즘에서는 실행가능성 검사를 통하여 불필요한 재시작 버전의 생성을 효과적으로 억제할 뿐만 아니라 높은 우선순위를 갖는 트랜잭션의 불필요한 실행을 방지하는 효과를 갖는다. 즉, 낮은 우선순위 트랜잭션이 실행되는 도중에 높은 우선순위를 갖는 트랜잭션이 해당 자원을 요청하는 상황에서, 2PL-HP 방법처럼 무조건 낮은 우선순위 트랜잭션을 재시작 시키거나, AVCC 방법처럼 항상 대기 버전과 재시작 버전을 동시에 유지하는 것이 아니라 높은 우선순위를 갖는 트랜잭션의 실행가능성 검사 결과에 따라 재시작 될 필요가 있을 때에만 낮은 우선순위 트랜잭션을 재시작 시키고, 그렇지 않은 상황이면 낮은 우선순위 트랜잭션의 계속적인 실행을 보장한다.

(3) 마감시간 초과 비율 비교

마지막 실험에서는 비교 대상인 AVCC와 제안하는 2PL-FT 알고리즘의 마감시간 초과 비율을 비교한다. 그림 3.12와 같이 초당 트랜잭션 도착율이 낮은 경우에는 두 방법 간의 성능 차이가 거의 없다. 반대로 초당 트랜잭션 도착율이 커지면서 제안하는 2PL-FT 알고리즘의 성능 향상이 두드러지게 나타난다. 트랜잭션의 마감시간 초과 비율에 대한 성능평가 결과 제안하는 알고리즘이 26.8% 만큼 더 낮게 나타났다. 그 이유는 다음과 같다. AVCC 방법은 충돌 발생 시 낮은 우선순위 트랜잭션을 무조건 철회시키지 않고 대기 버전과 이에 대한 재시작 버전을 생성시킨다. 그리고 높은 우선순위 트랜잭션이 마감시간 내에 정상적으로 완료되는지 여부에 따라 적당한 버전을 선택하여 낮은 우선순위 트랜잭션의 불필요한 철회 문제를 해결한다. 이와 달리 제안하는 2PL-FT 알고리즘은 높은 우선순위 트랜잭션의 실행가능성 검사를 통해 낮은 우선순위 트랜잭션의 불필요한 철회 문제 및 대기 문제 모두를 해결한다.



(그림 15) 트랜잭션 도착율에 따른 마감시간 초과 비율

제안하는 알고리즘에서는 실행가능성 검사를 위해 트랜잭션의 예상실행시간을 사용하므로 계산상의 오차 발생이 불가피하다. 예상실행시간은 해당 트랜잭션의 평균 실행시간과 최소실행시간의 중간값을 사용하므로 시스템 가동 초기에는 오차가 크지만 시스템이 정상적으로 가동되는 상황에서는 오차가 크게 감소하는 특징을 갖는다. 따라서 그림 15와 같이 초당 트랜잭션 도착율이 증가하면서 계산상의 오차 범위가 줄어들어 정확도가 높아지면서 높은 우선순위 트랜잭션의 불필요한 실행을 막고, 낮은 우선순위 트랜잭션의 불필요한 철회 및 대기 문제를 효율적으로 해결한다. 이에 따라 마감시간 초과 비율이 AVCC 방법에 비해 크게 감소되는 것을 알 수 있다.

또한 그림 15와 같은 성능 차이를 보이는 것은 AVCC 방법이 갖는 문제점과도 크게 관련이 있다. AVCC 방법은 두 번째 실험 결과에서와 같이 트랜잭션 재시작 비율이 초당 트랜잭션 도착율 증가에 따라 커지는 특성을 갖는다. 따라서 초당 트랜잭션 도착율이 커질수록 심각한 자원 낭비를 초래하게 되고, 결국 일반 트랜잭션들의 정상적인 실행을 방해하여 전체적인 시스템 성능을 떨어뜨린 것이라 할 수 있다.

5. 결 론

최근까지, 실시간 데이터베이스에 관한 수많은 연구에서 트랜잭션의 마감시간 초과 비율을 최소화하기 위한 목적으로 다양한 병행실행제어 알고리즘들이 제안되었다. 최근에 AVCC 방법이 제안되었다. 그러나 이 방법은 낮은 우선순위를 갖는 트랜잭션의 불필요한 철회 문제를 해결하기 위해 항상 두 개의 버전을 동시에 유지해야 하는 또 다른 문제점을 갖는다. 게다가 제 2, 제 3의 재시작 버전을 관리하기 위한 계층적 구조를 갖는 복잡한 버전 관리가 추가적으로 요구된다. 따라서 낮은 우선순위 트랜잭션의 불필요한 철회 및 불필요한 대기 문제를 모두 해결할 수 있는 보다 효율적인 방법이 요구된다.

이에 본 논문에서는 기존 연구에서 해결하지 못하고 있는 문제점들을 종합적으로 분석하고 이를 개선할 수 있는 새로운 그리고 효율적인 방안을 제시하였다. 본 논문에서는 낮은 우선순위 트랜잭션이 불필요하게 대기하거나 또는 철회되는 문제를 모두 해결할 수 있는 새로운 실시간 병행실행제어 알고리즘(2PL-FT)을 제안하였다. 성능평가 결과 AVCC 방법에 비해 전체 트랜잭션의 재시작 비율이 적고, 마감시간 초과 비율도 적은 것을 알 수 있었다. 그 이유는 충돌 발생 시, 높은 우선순위 트랜잭션에 대한 실행가능성 검사를 실행함으로써 다음과 같은 장점을 갖기 때문이다. 먼저, 높은 우선순위 트랜잭션의 불필요한 실행을 방지하여 자원의 낭비를 막는다. 그리고 낮은 우선순위 트랜잭션의 불필요한 철회 및 대기 문제를 해결함으로써 불필요한 자원의 낭비를 막고 대기 시간을 줄이는 효과를 얻는다. 따라서 이와 같은 장점들로 인해 전체적인 시스템의 성능을 향상시키는 것이다.

향후 연구과제는 제안하는 실시간 병행실행제어 알고리즘을 통합 환경에서 평가하여 그 우수성을 입증하는 것이다. 많은 기존 연구 결과에 의하면, 자원 경쟁이 심한 경우와 그렇지 않은 경우에 각각 비관적 병행실행제어 방법과 낙관적 병행실행제어 방법이 우수한 성능을 보인다. 따라서 펌, 소프트, 비 실시간 특성을 갖는 다양한 트랜잭션을 대상으로 여러 가지 형태와 크기를 갖는

멀티미디어 데이터에 대해 재 참조 빈도를 변화시키고, 자원 경쟁률을 다양하게 변화시켜 가면서 전체적인 시스템 성능의 변화에 대한 연구를 수행할 것이다.

참고문헌

[1] Ben Kao and Hector Garcia-Molina, "An Overview of Real-Time Database Systems," In Sang H. Son, editor, Advances in Real-Time Systems, chapter 19, Prentice Hall, 1995.

[2] Krithi Ramamritham, "Real-Time Databases," International Journal of Distributed and Parallel Databases, 1(2), 1993.

[3] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," IEEE Transactions on Knowledge and Data Engineering, 7(4), pp. 513~532, August 1995.

[4] N. Redding, "Network Services Databases," Proceedings of IEEE Global Telecommunications Conference, pp. 1336~1340, 1986.

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

[6] S. Hvasshovd and O. Torbjornsem, "The ClustRa Telecon Databases High Availability, High Throughput, and real-time response," Proceedings of 21th VLDB, pp. 469~477, 1995.

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

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

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

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

[11] Liu C L and Layland J W, "Scheduling Algorithms for Multi-programming in Hard real-time Environment," 1976.

[12] Stankovic J, Spuri M and Natale M D, "Implications of Classical Scheduling Results for Real-Time Systems," 1995.

[13] Sha L, Rajkumar R, Son S H and Chang C H, "A Real-Time Locking Protocol," 1991.

[14] Abbott R and Garcia-Molina H, "Scheduling Real-time Transactions: A Performance Evaluation," 1992.

[15] Haritsa J R, Carey M J and Livny M, "Data Access Scheduling in Firm Real-time Database Systems," 1992.

[16] Lee J, "Concurrency Control Algorithms for Real-time Database Systems," 1994.

[17] J. Haritsa, M. Carey and M. Livny, "On Being Optimistic About Real-Time Constraints," In Proceedings of the ACM Symposium on Principles of Database Systems, April 1990.

[18] Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions: A Performance Evaluation," Proceedings of the 14th Conference on Very Large Database Systems, Aug. 1988.

[19] Robbert Abbott and Hector Garcia Molina, "Scheduling real-time transactions with disk resident data," Proceedings of 15th VLDB, pp. 385~396, 1989.

[20] J. Stankovic and W. Zhao, "On real-time transaction," ACM SIGMOD Record 17, pp. 4~18, 1988.

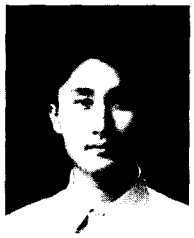
[21] J. Stankovic and K. Ramamrithmm, "What is predictability for real-time systems?," Real-Time Systems Journal, Vol. 2, No. 4, pp. 247~252, 1990.

[22] D. Hong, Sharma Chakravarthy and Theodore Johnson, "Locking Based Concurrency Control for Integrated Real-Time Database Systems," RTDB'96, pp. 138~143, 1996.

[23] Patrick E. O'Neil, Krithi Ramamritham and Calton Pu, "A Two-Phase Approach to Predictably Scheduling Real-Time Transactions," 1994.

[24] Yoshifumi Manabe and Shigemi Aoyagi, "A Feasibility Decision Algorithm for Rate Monotonic and Deadline Monotonic Scheduling," 1st Real-Time Technology and Applications Symposium, pp. 212-218, 1995.

◐ 저 자 소 개 ◑



이 석 재

2000년 2월 충북대학교 정보통신공학과 졸업(공학사)
2002년 2월 충북대학교 정보통신공학과 졸업(공학석사)
2002년 3월~현재 : 충북대학교 정보통신공학과 박사과정 재학
관심분야 : 메모리 상주형 데이터베이스 시스템, 실시간 시스템, 클러스터 저장시스템 등
E-mail : sjlee@netdb.chungbuk.ac.kr

박 새 미

2002년 2월 충북대학교 컴퓨터공학과 졸업(공학사)
2002년 3월~현재 : 충북대학교 정보통신공학과 석사과정 재학
관심분야 : 데이터베이스 시스템, 파일 시스템, XML 등
E-mail : prettysam@netdb.chungbuk.ac.kr

강 태 호

1999년 2월 호원대학교 정보통신공학과 졸업(공학사)
2002년 8월 충북대학교 정보산업공학과 졸업(공학석사)
2003년 2월~현재 : 충북대학교 정보통신공학과 박사과정 재학
관심분야 : 데이터베이스 시스템, 자료저장 시스템, 웹 컨텐츠 관리 시스템, 웹 마이닝 등
E-mail : segi21@netdb.chungbuk.ac.kr

유 재 수

1989년 2월 전북대학교 공과대학 컴퓨터공학과 졸업(공학사)
1991년 2월 한국과학기술원 전산학과 졸업(공학석사)
1995년 2월 한국과학기술원 전산학과 졸업(공학박사)
1995년~1996년 목포대학교 전산통계학과 전임강사
1996년~현재 : 충북대학교 전기전자컴퓨터공학부 부교수
관심분야 : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스, 분산 객체 컴퓨팅 등
E-mail : yjs@cbucc.chungbuk.ac.kr