네트워크 및 서비스

일반논문
효율적인 네트워크 자원 관리를 위한 호스트의 접속 유형 판별에 관한 연구

허 민, 김영섭 965

이중망에서 MPCTP를 위한 패킷 스케줄링 방법과 수신단 기반의 손실 복구 방법

오봉환, 김한야, 이재용 975

ID-LOC 분리 기반 인터넷 구조에서 분산형 메핑 시스템의 구현 및 평가

강형우, 최낙종, 김지인, 정희영, 고석주 984

이기종 무선 네트워크에서 대중연결을 이용한 전송을 제어

정현진, 최승식 993

uPaging : 실시간 위치 인식 기반의 음성매시지 전송 시스템

박우진, 전상호, 강준호 1004

IEEE 802.11s기반 무선 매쉬 네트워크에서 네트워크 코딩 적용을 위한 고려사항 분석

이규환, 조성현, 김재현 1014

Bio-inspired 알고리즘을 이용한 OFDMA 기반 매쉬 네트워크의 분산 주파수 동기화 기법

유현준, 이미나, 조용운 1022

공군 전장 환경에서 효과적 주파수 관리를 위한 Spectrum COP 설계 제안

구자열 1033

시스템 성능 및 비용 트래픽에 대한 트래픽서널 메모리의 층중 관리 정책 영향 분석

김영규, 문병인 1041

Super Wi-Fi 환경에서 서비스 연속성에 관한 끌김없는 채널이동 방안 연구

김영우, 전유진, 박상원, 박진우 1050

2012 하계학술대회 특집

ICT 생태계 구축을 위한 기업 전략 분석 및 정책 제안

유지은, 이기백, 최문기, 조항정 1058

Ka 대역 위성 쏟각 전력 제어 기술 연구

신동현, 윤소현, 문성모, 이홍열, 엘만석, 염인복 1072

DNS 트래픽 기반의 사이버 위협 도메인 탐지

임성희, 김종현, 이병길 1082
Analysis of the Influence of the Conflict Management Policy of the Transactional Memory on the System Performance and Bus Traffic

Young-Kyu Kim*, Byungin Moon*

ABSTRACT

The transactional memory was proposed to solve the problems of the conventional lock-based synchronization methods in the shared memory multiprocessor system. Various implementation methods for putting the high performance transactional memory to practical use have been continuously studied. However, these studies focus only on the commercialization and performance enhancement of the transactional memory. Besides, there have been few studies to analyze the system overhead of the transactional memory according to the conflict management policy. Thus this paper classifies hardware transactional memory, which is one kind of transactional memories, into four types according to the conflict management policy, and then compares and analyzes their performance and system bus traffic through their modeling and simulation. In addition, the most effective conflict management policy for the hardware transactional memory is presented through these comparison and analysis.

* 본 논문은 경북대-삼성전자 반도체 산학협력단협회 연구과제에 의해 지원된 연구 결과입니다.
* 주저자: 경북대학교 전자공학과 박사, kyk79@ece.knu.ac.kr
* 교신저자: 경북대학교 IT대학 전자공학부, bhmoon@knu.ac.kr, 중형회의

Key Words: Transactional memory, multiprocessor, synchronization, shared memory multiprocessor
I. 서론

발열 및 클럭 스케우(clock skew) 등의 문제로 인해 싱글 코어 프로세서의 발전은 한계에 이르렀고, 인텔, AMD 등의 프로세서 제조사들은 고성능 프로세서에 대한 시그널 요구에 부응하기 위해 여러 허브 혹은 그 이상의 프로세서를 적절한 멀티 코어 프로세서를 개발하여 출시하였다. 이는 결과로 인해 하드웨어 발전에 의존하여 수동적인 입장을 취했던 소프트웨어 또한 하드웨어 자원을 충분히 활용하기 위한 멀티 코어 기반으로 프로그램의 처리가 변화하였다. 그러나 순차적인 처리 방식에 의존해진 프로그래머들은 새로운 프로그램의 처리에 효율적으로 대처하지 못하였고, 이는 멀티프로세서 시스템의 제 성능을 이끌어 내지 못하

로 주요 원인으로 지적되었다. 특히 발생 여부가 결정되지 않은 시스템을 예측하여 프로그래밍 해야 하는 멀티 처리 프로그램에서 lock을 사용하는 프로세서 시스템의 제약은 멀티 프로그램을 더욱 낮춰야 하는 요소로 작용하였다.10

한편 멀티프로세서 시스템은 프로세서들과 메모리의 구조 및 공유 방법에 따라 분산메모리(distribute memory) 구조와 공유메모리(shared memory) 구조로 분류된다. 멀티프로세서 시스템은 구조에 따라 시스템의 성능은 크게 달라진다. 2004년을 기점으로 출시되었던 싱글 코어 기반의 멀티프로세서들은 대부분 공유메모리 멀티프로세서(shared memory multiprocessor)로 유사한 구조로 설계되었다.11

그림 1. 공유메모리 멀티프로세서 구조
Fig. 1. Architecture of a shared memory multiprocessor

공유메모리 멀티프로세서 시스템은 그림 1과 같이 하나의 공유메모리가 시스템 버스를 통하여 여러 프로세서들에게 공유되어지는 구조로서, 각각의 프로세서들은 동시에 병렬적 공유메모리의 접근을 요구하기 때문에 프로세서들 간의 동기화는 항상 유지되어야 한다. 하지만 공유메모리 멀티프로세서 시스템에서 lock을 사용하는 전통적인 블로킹(blocking) 기반 프로세서의 동기화 기법들은 멀티프로세서의 병렬성을 저해하고 대중처리 프로그래밍을 어렵게 할 뿐만 아니라 데이터(deadlock), 콘보징(convoying), 우선순위 역전(priority inversion) 등과 같은 시스템에 치명적인 결과를 초래하는 문제점을 겪게 된다. 이러한 문제점들은 해결하기 위한 대안으로 트랜잭션 멀티프로세서가 제안되었다. 트랜잭션 멀티프로세서는 기존의 케시 일관성 프로토콜을 일부 수정하여 구현한 브로크링(non-blocking) 방식의 프로세서 동기화 방식으로, lock을 사용하는 전통적인 동기화 방식의 문제점들은 모두 해결할 수 있기 때문에 이를 실험하기 위한 연구가 활발히 진행되고 있다.12 트랜잭션 멀티프로세서는 프로세서들 간의 동작 정도와 총괄 관리 정책에 따라 트랜잭션 수행 속도 및 시스템 버스의 트래픽 정도가 크게 달라진다. 특히 시스템 버스는 공유메모리 멀티프로세서 시스템의 성능과 구조를 결정짓는 매우 중요한 요소이며, 시스템 버스의 성능은 시스템 버스와 트래픽 관리 방법에 의해 크게 영향을 받는다.13

이에 본 논문은 트랜잭션 멀티프로세서의 큰 분류 중 하나인 하드웨어 트랜잭션 멀티프로세서의 트랜잭션 동작을 축약 관리 정책에 따라 나가지 구분하여 각각 모델링하고, 프로세서들 간의 동작 정책에 따른 시스템의 성능과 시스템 버스의 트래픽 정도를 시뮬레이션을 통해 비교 분석한다. 그리고 분석한 시뮬레이션 결과를 통하여 시스템의 성능 향상에 가장 크게 기여할 수 있는 효율적인 축약 관리 정책을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 트랜잭션 멀티프로세서의 개념과 트랜잭션 멀티프로세서의 총괄 관리 정책에 대해 설명하고 3장과 4장에서는 모델링 및 시뮬레이션 결과에 대해 설명한다. 마지막으로 5장에서는 본 논문의 결론을 밟는다.

II. 관련 연구

2.1. 트랜잭션 멀티프로세서

lock을 사용하는 블로킹 기반 프로세서 동기화 방식은 공유메모리에 접근하기 위한 선취권을 요구하는 방식으로서, 선취권을 획득한 프로세서에게만
임계역에 접근하는 것을 허용하며 그렇지 못한 프로세스들은 불록(block) 처리한다. 불록 처리된 프로세스들은 선취권을 부여 받기 때문에 작업을 중단하고 대기해야 한다. 이러한 불록징 방식의 프로세스 동기화는 멀티프로세스 시스템의 병렬적인 동작을 조조리고 순차적인 동작을 요구함으로써 시스템의 성능 저하를 이기한다. 트랜잭션 페포리는 lock을 사용하는 불록징 기반 동기화 방식의 한계를 극복할 수 있는 프로세스 동기화 방식으로, 여러 프로세스들이 공유메모리에 접근할 때 중동이 발생하지 않는 경우에 한해서 병렬적인 동작을 보장하는 불로킹 기반의 기술이이다.

트랜잭션 페포리의 동작은 트랜잭션(transaction) 단위로 수행된다. 트랜잭션이라면 일련의 load/store 명령들이 하나의 명령이라고 간주하여(atomically)로 수행되어야하는 명령가 그룹이다. 트랜잭션 페포리는 공유메모리에 대한 선취권을 요구하지 않는 동기화 방식으로, 프로세스들이 공유메모리에 접근할 때 우선 트랜잭션을 수행한 후 다른 프로세스들의 중동 여부를 검사한다. 중동 검사에서 중동되는 경우에 없다면 트랜잭션은 완료되지만, 만약 중동이 감지된다면 수행했던 트랜잭션을 취소하고 공유메모리를 트랜잭션 수행 전의 값으로 복구한 후 다시 처음부터 트랜잭션을 시도한다. 여기서, 트랜잭션을 취소하고 데이터를 복구하는 동작을 abort라고 하고, 다른 프로세스와 중동 없이 트랜잭션을 완료하는 것을 commit이라고 한다.

그림 2는 프로세스 P0, P1, P2가 공유메모리의 공유메모리 A, B, C, D에 접근할 때, lock을 사용하는 불로킹 기반 방식과 트랜잭션 페포리의 동작 과정을 비교 설명하는 그림이다. 트랜잭션의 경우, 프로세스들은 접근하는 공유메모리에 중동이 발생하지 않으면서 중동을 감지한 경우에 트랜잭션을 수행하여 commit에 도달한다. 하지만 P2가 공유메모리 C에 중동을 감지한 경우에는 수행했던 트랜잭션을 abort하고 다시 트랜잭션을 시도하여 commit에 성공한다. Lock의 경우에는 동일 공유메모리에 대한 동시 접근과 무관하게 lock을 소유한 프로세스에게만 제어력 영역의 진입이 허용되며 lock을 위해 대기하는 프로세스들의 작업은 순차적으로 처리된다. 즉, 동일한 환경에서 동일한 작업을 수행하더라도 중동 가능한 작업을 병렬적으로 처리한 트랜잭션 페포리의 경우에 lock을 사용하는 순차적인 처리 방식인 불로킹 기반 방식보다 모든 작업이 완료되는 시점이 다음 앞서는 것일을 알 수 있다.

그림 2. Lock과 트랜잭션의 비교.

Fig. 2. Comparison of locks and transactions.

이러한 트랜잭션 페포리는 논리적으로 간단한 병렬 동화 방식으로 프로세스들 간의 병렬적인 동작을 더욱 보장해 줄 뿐만 아니라 불로킹 기반의 기술들에 이 기여하는 데다가 더 큰 평판이 들을 같은 작업을 배제할 수 있고, 프로세스들 간의 동기화를 트랜잭션 페포리에서 관리함으로 인해 병렬 프로그래밍을 더욱 쉽게 해주는 장점을 가져온다.

트랜잭션 페포리는 트랜잭션을 구성하는 방법에 따라 소프트웨어 트랜잭션 페포리, 하드웨어 트랜잭션 페포리, 하이브리드(hybrid) 트랜잭션 페포리 등 크게 세 가지로 구분된다. 소프트웨어로 구현된 소프트웨어 트랜잭션 페포리는 프로세스 간의 다양한 경험 상황에 대한 유연한 대처가 가능하지만 모든 처리 방식이 소프트웨어에 의해 수행되므로 하드웨어 트랜잭션 페포리는 비해 오버헤드가 매우 큰 단점이 있다. 하드웨어 트랜잭션 페포리는 트랜잭션 수행 속도는 높아지지만 한정된 크기의 트랜잭션만 처리 가능하고 다양한 경험 상황에 대처하기 힘든 단점이 있다. 소프트웨어와 하드웨어를 융합시킨 형태의 하이브리드 트랜잭션 페포리는 소프트웨어와 하드웨어의 장점을 모두 겸비하는 것이 실제 성능은 하드웨어 트랜잭션 페포리에 비해 매우 저조한 특성을 보였다. 하이브리드 트랜잭션 페포리의 성능은 기대했던 수준으로 안정적이기 위해서는 소프트웨어에서 비롯되는 오버헤드를 크게 줄이려는 데에는 적을 해결할 수 없는 문제이다. 그러므로 하이브리드 트랜잭션 페포리의 성능 향상을 위해서는 하드웨어에
의 지원이 결실한 상징이며, 이에 따라 하드웨어 트랜잭션 메모리의 다양한 관리 전략에 대한 연구는 더욱 중요해지고 있다[8,9]. 이러한 이유로, 본 논문에서는 동일한 성능 상황에서 하드웨어 트랜잭션 메모리의 축출 관리 정책에 따른 성능과 시스템 오버헤드를 비교 분석하기 위하여 하드웨어 트랜잭션 메모리를 모델링하여 연구를 진행하였다.

2.2. 트랜잭션의 축출 관리 정책

동시에 수행되는 트랜잭션들 간의 충돌 여부를 관리해야 하며 충돌이 발생했을 경우 이 미 수행했던 작업을 취소하고 데이터를 복구하는 등의 작업이 필요하다. 이와 같이 충돌이 감지되었을 때 수행되는 후처리 작업의 세부적인 동작은 충돌 관리 정책에 의해 결정되며 이는 트랜잭션 메모리의 특성과 결정되는 가장 중요한 요소로, 충돌 관리 정책은 충돌 감지(conflict detection) 정책과 데이터 관리(version management) 정책으로 구분되며, 각각은 동작 방식에 따라 eager 방식과 lazy 방식으로 분류된다[10].

충돌 감지 정책은 트랜잭션을 시도한 공유데이터의 일관성이 떨어졌음을 검사하는 정책으로 eager 방식과 lazy 방식으로 구분된다. eager 방식의 충돌 감지 정책은 트랜잭션을 수행하는 과정에서 항상 충돌 발생 여부를 검사하여 충돌이 발생하는 즉시 트랜잭션을 중단하고 데이터를 트랜잭션 이전 상태로 복구한 후 처음부터 다시 트랜잭션을 수행한다. 이러한 경우 대기 방식은 충돌이 발생하는 즉시 감지가 가능하므로 프로세서들의 간의 경합 정도가 높은 환경에서 효율적인 것으로 알려져 있다. 반면, lazy 충돌 감지 정책은 트랜잭션을 수행한 후 commit 지점에서 충돌 발생 여부를 검사하는 방식으로 충돌을 감지하는 시간에 eager 방식보다 느리지만 충돌 감지를 위한 오버헤드가 eager 방식보다 상대적으로 낮기 때문에 프로세서들 간의 경합이 치열하지 않은 환경에 효율적이며[11,12].

데이터 관리 정책은 쓰기 동작이 수행되는 트랜잭션에서 충돌이 감지되었을 때 원본 데이터를 복구하는 방법에 대한 정책으로 이 역시 eager 방식과 lazy 방식으로 구분된다. eager 데이터 관리 정책은 트랜잭션 과정에서 쓰기 동작을 수행할 때 해당 영역에 즉시 쓰기를 수행하고 원본 데이터는 바이에 보존해 두는 방식이다. 이후 트랜잭션이 abort 되면 바이에 저장해두고 원본으로 공유데이터를 복구하고, 트랜잭션이 commit되면 추가적인 작업 없이 트랜잭션을 완료한다. Eager 데이터 관리 정책의 경우, 발생 시 비과 공유영역의 데이터 이동에 대한 오버헤드가 발생하지만, 트랜잭션이 완료된 후 데이터이동을 수행하지 않고 버퍼에만 쓰기 동작을 수행한다. 이 후 트랜잭션을 commit 할 때 비과의 내용을 해당영역에 업 데이터 한다. 만약 트랜잭션 과정에서 충돌이 발생한다면 수행했던 트랜잭션은 취소되지만 데이터 복구를 위한 추가적인 작업 없이 트랜잭션을 다시 시작한다. Lazy 데이터 관리 정책은 항상 일정한 수준의 오버헤드를 유지하므로 프로세서들 간의 경합이 높은 환경에 유용한 정책이다[11,12].

이러한 트랜잭션 메모리의 축출 관리 정책들은 시스템 버스의 속도와 품질, 그리고 스레드 간 통신의 복잡성 등에 보다 시스템 버스 에 의해 동작하기 때문에 시스템 버스의 성능이 시스템의 성능과 직접적으로 관련된다. 트랜잭션 메모리 메모리 버퍼와 공유데이터를 위한 버퍼에 데이터를 저장하고 트랜잭션은 시스템 버스를 기반으로 동작한다. 그러므로 트랜잭션 메모리 메모리 모델을 설계 시에는 충돌 감지 정책 및 메모리 관리 정책에 따른 트랜잭션 메모리의 성능뿐만 아니라, 트랜잭션 동작 중 발생할 시스템 버스의 트래픽 역시 충분히 고려되어야 한다. 이에 본 논문은 충돌 관리 정책에 따른 트랜잭션 메모리의 성능과 시스템 버스의 트래픽 정도를 분석하기 위하여 하드웨어 트랜잭션 메모리 메모리의 축출 관리 정책에 따라 각각 모델링하였다.

III. 하드웨어 트랜잭션 메모리 모델링

하드웨어 트랜잭션 메모리의 축출 관리 정책에 따른 성능과 시스템 오버헤드를 분석하기 위한 모델링은 C 언어를 사용하여 구현하였다. 모델링된 하드웨어 트랜잭션 메모리의 구조는 그림 3과 같다.
그림 3. 하드웨어 트랜잭션 메모리 시스템 모델의 구조

Fig. 3. Architecture of the modeled hardware transactional memory system


이와 같은 모델링은 트랜잭션 메모리의 총돌 관리 정책에 따라 네 가지로 각각 구현하였다. 4 가지 중돌 관리 정책은 총돌 감지 정책과 데이터 관리 정책에 따라 lazy_clean, lazy_eager, eager_clean, eager_eager로 구분되며, 각각의 모델링에 적용된 동작 방식은 그림 4의 순서도와 같다.

그림 4. 총돌 관리 정책에 따른 트랜잭션 동작 순서도: (a) lazy 총돌 감지, lazy 데이터 관리, (b) lazy 총돌 감지, eager 데이터 관리, (c) eager 총돌 감지, lazy 데이터 관리, (d) eager 총돌 감지, eager 데이터 관리

Fig. 4. Flowcharts of transaction operations according to the conflict management policy: (a) lazy conflict detection and lazy version management, (b) lazy conflict detection and eager version management, (c) eager conflict detection and lazy version management, and (d) eager conflict detection and eager version management

그림 4(a) 및 4(b)의 lazy 총돌 감지 정책은 트랜잭션의 commit을 수행하기 전에만 총돌 여부를 감지하고, 총돌이 감지되면 실행 중인 트랜잭션을 무시하고 처음부터 다시 트랜잭션을 수행한다. 그림 4(c) 및 4(d)의 eager 총돌 감지 정책은 트랜잭션을 수행하는 과정에서 항상 총돌 여부를 감지하여 총돌이 감지되면 실행 중인 트랜잭션을 무시하고 처음부터 다시 트랜잭션을 시도한다. 그림 4(a) 및 4(c)의 eager 데이터 관리 정책은 임의의 영역에 트랜잭션을 수행 한 후 트랜잭션을 commit 할 때만 레플리 캐시의 해당 영역에 결과를 업데이트 한다. 그림 4(b) 및 4(d)의 eager 데이터 관리 정책은 트랜잭션 수행 과정에서 레플리 캐시의 해당 영역에 작업 결과를 무시 작성하고, 만약 트랜잭션이 commit되어링 복사체들의 원본 데이터를 이용하여 해당 영역의 데이터를 복구한다.

트랜잭션 수행 도중 발생하는 캐시메모리 및 공유메모리 간의 동작은 모델링의 간소화를 위하여 세부적인 구현은 생략하였으며 선택 조사된 연구들에서 사용된 서言った의 설정 값을 사용하여 표 1의 elem 값과 같이 설정을 설정하고 적용하였다[14-16]. 단, 본 연구의 모델링 구현에서는 설정할 수 있는 최소 자료 시간은 1 nsec인 값을 감안하여
지연 시간이 가장 짧은 메시지모드의 지연 시간을 기준으로 참조한 설정 값을 동일한 비율로 조절 하였다. 캐시 사상 방식은 직접 사상 방식으로 설정 하였으며 버스 중개 방식은 in-order 방식으로 구현 하였다. 총 몇 가지 동작은 데이터 복구가 필요한 총 몇과 데이터 복구가 필요하지 않은 총 몇으로 구분하 여 측정하였으며, 총 몇 검사를 위해서는 트랜잭션 캐시의 상대 비트를 읽기 위한 접근 시간이 요구되 다. 데이터 백업을 위한 비트는 SRAM으로 가정하 여 구현하였기 때문에 캐시에러와 동일한 접근시간을 적용하였다. 이러한 설정들을 통해 전체 시스템 수행 시간은 Windows API 에서 지원하는 QueryPerformanceCounter() 함수를 사용하여 32 개의 프로세서들이 동작을 시작하는 시점에서 32개 의 프로세서들이 주어진 모든 동작을 완료하는 시 점까지의 시간을 측정하였다.

표 1. 사용된 시스템 파라미터

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processor</td>
<td>32 processors</td>
</tr>
<tr>
<td>Regular Cache</td>
<td>SRAM, 1 ns access latency</td>
</tr>
<tr>
<td>Transaction Cache</td>
<td>SRAM, 1 ns access latency</td>
</tr>
<tr>
<td>Shard Memory</td>
<td>DDR2, 25 ns access latency</td>
</tr>
<tr>
<td>System bus</td>
<td>5 ns latency</td>
</tr>
</tbody>
</table>

IV. 실험 결과

일상에서 설명한 모델링을 통하여 네 가지 트랜잭션 메모리 모델들 각각의 트랜잭션 수행 시간과 시스템 버스 요청 횟수를 측정하는 실험을 하였 다. 트랜잭션 수행 시간은 32개의 프로세서들 각각 이 64개의 트랜잭션을 모두 성공하는데 소요된 평 균 시간으로, 접근 가능한 공유메모리의 개수를 변화시키면서 경향 강도가 높은 경우에 대해야 낮은 경 우 모 두를 대상으로 측정하였다. 트랜잭션 수행시간 은 프로세서가 트랜잭션을 완료하는데 소요되는 시 간만을 대상으로 하였으며 트랜잭션의 commit으로 인한 공유메모리 업데이트 동작을 측정시간에서 제외하였다. 시스템 버스 요청 횟수는 32개의 프로세서 각각이 64개의 트랜잭션을 모두 성공하기까지 시스템 버스를 요청한 평균 횟수를 측정한 것으로, 트랜잭션 수행시간 측정시간에서 각각 접근 가능한 공유메모리 개수를 변화시키면서 측정하였다. 측정은 16회 반복하여 측정하였으며, 대회마다 접근 가능한 공유메모리의 개수를 2의 배수로 증가하여 측정하였다. 프로세서들 간의 경합 정도가 난수에 의해 결정되는 점을 감안하여 다음 각각의 있는 실험결과를 위해 동일한 실험을 반복하여 얻은 데이터의 평균값을 분석에 사용하였다.

그림 5. 접근 가능한 공유메모리 개수에 따른 평균 트랜잭션 수행 시간

Fig. 5. Average transaction processing time according to the number of shared addresses accessible by processors

그림 6. 접근 가능한 공유메모리 개수에 따른 시스템 버스 평균 요청 횟수

Fig. 6. Average number of system bus requests according to the number of shared addresses accessible by processors
실험 결과는 그림 5, 그림 6 및 그림 7과 같다. 그림 5는 32개의 프로세서들 각각이 64회의 트랜잭션을 수행하는데 소비된 시간을 측정한 결과로 프로세서들의 성능 차이에 따라 각각의 충돌 감지 정책들에 대한 트랜잭션 전체 수행 시간을 나타내고 있다. 그림 6은 32개의 프로세서들 각각이 64회의 트랜잭션을 완료하기 위해 시스템 버스를 요청했던 횟수를 측정한 결과이다. 그림 7은 32개의 프로세서들 각각이 64회의 트랜잭션을 수행하는 동안 감지했던 충돌 횟수를 측정한 결과이다.

Lazy 충돌 감지 정책과 lazy 데이터 관리 정책을 사용한 모델(lazy_lazy)은 트랜잭션 수행 시간 및 시스템 버스 요청 전단에서 프로세서의 성능 차이에 따른 영향을 가장 적게 받는 것으로 분석된다. 이 모델은 트랜잭션 수행 중 commit 직전에 충돌 감지를 실행하므로 트랜잭션 최종의 불필요한 동작이 최소화되고 트랜잭션들의 병렬적인 흐름을 가능할 수 있으며[6,12], commit 시에만 공유영역에 트랜잭션 결과를 업데이트하기 때문에 트랜잭션 결과가 abort될 경우 원본 데이터를 복원하기 위한 시스템 버스 트래픽이 발생하지 않는다. 그러므로 트랜잭션 수행시간을 측정한 실험과 시스템 버스 요청 횟수를 측정한 실험에서 프로세서의 성능 차이에 걸쳐 다른 모델들에 비해 가장 작은 성능을 보였다. 또한 이러한 충돌 감지 정책은 채도로 되는 트랜잭션 흐름을 최소화하기 때문에 트랜잭션 충돌 감지 횟수를 측정한 실험에서도 다른 모델들에 비해 낮은 수치를 기록하였다.

Lazy 충돌 감지 정책과 eager 데이터 관리 정책을 사용한 모델(eager_eager)은 프로세서 간의 경쟁이 높은 구간에서는 lazy_lazy 모델의 트랜잭션 수행 시간에 비해 측정된 수행 시간이 높게 측정되었으며, 경쟁이 높은 구간에서는 lazy_lazy 모델의 트랜잭션 수행 시간과 비교적 가까운 값들이 측정되었다. 그리고 시스템 버스 요청 횟수의 경우 경쟁이 높은 구간에서 lazyLazy 모델과 큰 차이를 보였으며, 트랜잭션 충돌 감지 횟수에서도 lazy_lazy보다 더욱 많은 충돌을 감지한 것으로 나타났다. 이러한 결과는 eager 방식의 데이터 관리 정책이 공유데이터에 대한 입자트 동작을 lazy 방식보다 더 많이 수행하기 때문에 이에 따른 트랜잭션 간의 충돌 감지 횟수가 증가하기 때문이다. 또한 충돌 발생 시 데이터 복구를 위한 오버헤드가 발생하기 때문에 시스템 버스 요청 횟수도 lazy_lazy 모델보다 상대적으로 높게 측정되었다.

그림 8는,Eager_Lazy 모델에서 측정되는 트랜잭션 충돌 Fig. 8. Transaction conflicts detected in the Eager_Lazy model

그림 9는 Eager_Eager 모델에서 측정되는 트랜잭션 충돌 Fig. 9. Transaction conflicts detected in the Eager_Eager model
그림 8과 그림 9는 eager 총출 감지 정책을 사용하는 모델들의 시뮬레이션 수행 중 감지된 총출의 특성을 나타낸 그림이다. Eager 총출 감지 정책을 사용하는 모델의 경우 commit 수행 과정에서 감지되는 총출뿐만 아니라 commit 수행 이전의 단계인 트랜잭션 수행 과정에서도 총출이 감지될 수 있다.

Eager 총출 감지 정책과 lazy 데이터 관리 정책을 사용한 모델(eager, lazy)은 eager 총출 감지 정책을 사용하는 모델들에 비해 트랜잭션 수행 시간이 높게 측정되었다. 트랜잭션 총출 감지 횟수는 lazy, eager 모델의 트랜잭션 총출 감지 횟수보다 높게 측정되었다. 즉 시스템 버스 요청 횟수는 lazy, eager 모델의 시스템 버스 요청 횟수보다 낮게 측정되었다. 이러한 결과의 원인은, 총출이 발생 하더라도 즉시 총출을 감지하지 못하고 commit 시점까지 트랜잭션을 진행한 후에 데이터 복구 작업을 하는 lazy, eager 모델보다 다르게, eager, lazy 모델은 그림 8과 같이의 트랜잭션 수행 과정에서 총출을 감지함으로써 불필요한 시스템 버스 요청이 줄어들고 데이터 복구로 인한 버스 트래픽이 적기 때문이다. 하지만 eager 방식의 총출 감지로 인해 트랜잭션들간의 병렬적인 총출이 가중되고, 트랜잭션 과정에서 항상 총출을 감지해야 하는 추가적인 수행시간이 요구된다.

Eager 총출 감지 정책과 eager 데이터 관리 정책을 사용한 모델(eager, eager)은 프로세스들의 연계가 높은 구간에서 다른 모델들에 비해 트랜잭션 수행 시간과 시스템 버스 요청 횟수가 특히 높게 측정되었다. 이러한 결과는 eager 총출 감지 정책으로 인해 트랜잭션 과정에서 총출 감지에 반응하는 오버헤드가 증가하고 트랜잭션들의 병렬적인 총출이 증가하기 때문이다. 또한 eager 데이터 관리 정책으로 인해 콘텐츠 마이그레이션 빈도가 증가하기 때문에 그림 9와 같이 전체적인 총출 감지 횟수가 다른 모델들에 비해 증가하였고 데이터 복구가 요구되는 총출 감지 비용도 높아졌다.

본 장에서는 트랜잭션 데이터의 총출 관리 정책에 따른 데이터의 특성을 시뮬레이션 한 결과를 통해 각 모델들의 특성을 비교 분석하였다. 각 모델들의 성능과 특성을 분석한 본 결과, 프로세스들의 다양한 성능, 특성에 대한 트랜잭션 수행 시간 및 시스템 버스 트래픽 용량 정도를 모두 고려하였을 때, lazy 총출 감지 정책과 lazy 데이터 관리 정책을 사용한 모델이 최적화된 트랜잭션 동작을 통해 시스템의 성능 향상에 기여하는 정도가 가장 우수한 것으로 분석된다.

V. 결론
본 논문에서는 트랜잭션 데이터의 총출 관리 정책이 시스템 성능 및 시스템 버스 오버헤드에 미치는 영향을 알아보기 위하여, 총출 관리 정책을 두 가지로 분류하고 각각에 대한 하드웨어 트랜잭션 예측을 모델링하고 시뮬레이션 하였다. 시뮬레이션 결과 총출 감지 정책이 데이터 관리 정책을 비해 시스템의 성능에 큰 영향을 미치는 것으로 분석되며, 시스템 버스 오버헤드 측면에서는 데이터 관리 정책이 총출 감지 정책에 비해 미치는 영향이 큰 것으로 분석되었다. 총출 감지 정책의 경우 eager 방식보다 lazy 방식이 트랜잭션 수행 시간에서 더욱 좋은 성능을 보였으며, 데이터 관리 정책의 경우에도 eager 방식보다 lazy 방식이 더욱 효율적인 것으로 나타났다. 총출 감지 정책에 따른 데이터 모델들의 시뮬레이션 결과, lazy 총출 감지 정책과 lazy 데이터 관리 정책을 사용하는 것이 트랜잭션 수행 시간 및 시스템 버스 요청 횟수 측면에서 프로세스간의 경합 정도 방면에서 가장 효율적인 정책임을 알 수 있었다. 단, 이러한 결과는 각 프로세스들이 동일한 비율로 임의의 영역을 접근하여 동일한 작업을 수행한다는 가정에서 도출된 결과이기 때문에 제시된 결론의 일반화를 위해서는 다양한 변수들을 시뮬레이션에 적용하는 연구가 필요하다.

참고 문헌


김영규 (Young-Kyu Kim)
2005년 2월 경주대학교 전자공학과 학사 졸업
2011년 2월 경북대학교 모바일통신공학과 석사 졸업
2011년 3월 현대 정부대학교 전자기기컴퓨터학부 박사과정

문병인 (Byungin Moon)
1995년 2월 연세대학교 전자공학과 학사 졸업
1997년 2월 연세대학교 전자공학과 석사 졸업
2002년 2월 연세대학교 전기전자공학과 박사 졸업
2002년-2004년 하이닉스 반도체 설립연구원
2004년-2005년 연세대학교 연구교수
2005년-현재 경북대학교 IT대학 전자공학부 부교수

<관심분야> SoC, 컴퓨터 구조, 운영체제

<관심분야> SoC, 디지털 VLSI, 컴퓨터 구조

1049