논문 12-37B-11-09
한국통신학회논문지 '12-11 Vol.37B No.11
http://dx.doi.org/10.7840/kics.2012.37B.11.1041

시스템 성능 및 버스 트래픽에 대한 트랜잭셔널 메모리의 충돌 관리 정책 영향 분석

김 영 규•, 문 병 인°

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°

요 약

공유메모리 멀티프로세서 시스템에서, lock을 사용하는 전통적인 동기화 방식의 문제점들을 극복하기 위하여 트랜잭셔널 메모리(transactional memory)가 제안되었고, 고성능 트랜잭셔널 메모리를 실용화하기 위한 다양한 구현 방법들이 계속해서 연구되고 있다. 하지만 이러한 연구들은 트랜잭셔널 메모리의 실용화 및 수행 속도 개선에 주력하고 있으며, 충돌 관리 정책(conflict management policy)에 따른 트랜잭셔널 메모리의 시스템 오버헤드를 분석하는 연구는 부족한 실정이다. 이에 본 논문은 트랜잭셔널 메모리의 한 종류인 하드웨어 트랜잭셔널 메모리를 충돌 관리 정책에 따라 네 가지로 분류하고, 모델링과 시뮬레이션을 통해 이 네 가지의 성능과 시스템 버스 트래픽을 비교 분석한다. 그리고 이러한 비교 분석 결과를 바탕으로 시스템 성능에 가장 크게 기여 할 수 있는 효율적인 충돌 관리 정책을 제시한다.

Key Words : Transactional memory, multiprocessor, synchronization, shared memory multiprocessor

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 산학협력위원회 연구과제에 의해 지원된 연구 결과입니다.
• 주저자 : 경북대학교 전자전기컴퓨터학부, kyk79@ee.knu.ac.kr
° 교신저자 : 경북대학교 IT대학 전자공학부, bihmoon@knu.ac.kr, 종신회원
논문번호 : KICS2012-04-205, 접수일자 : 2012년 4월 10일, 최종논문접수일자 : 2012년 11월 6일

Ⅰ. 서 론

발열 및 클럭 스큐(clock skew) 등의 문제로 인해 싱글 코어 프로세서의 발전은 한계에 이르렀고, 인텔, AMD 등의 프로세서 제조사들은 고성능 프로세서에 대한 시대적 요구에 부응하기 위해 하나의 칩에 두 개 혹은 그 이상의 프로세서를 집적한 멀티 코어 프로세서들을 개발하여 출시하였다[1]. 이러한 결과로 인해 하드웨어 발전에 의존하여 수동적인 입장을 취했던 소프트웨어 또한 하드웨어 자원을 충분히 활용하기 위한 병렬 처리 기반으로 프로그램 패러다임이 변화하였다. 그러나 순차적인 처리 방식에 익숙해진 프로그래머들은 새로운 프로그램 패러다임에 효율적으로 대처하지 못하였고, 이는 멀티프로세서 시스템이 제 성능을 이끌어 내지 못하는 주요 원인으로 지적되었다. 특히 발생 여부가 결정되지 않은 사건들을 예측하며 프로그래밍 해야 하는 병렬 처리 프로그램에서 lock을 사용하는 프로세서 동기화 기법들은 병렬 프로그램을 더욱 난해하게 하는 요소로 작용하였다[2].

한편 멀티프로세서 시스템은 프로세서들과 메모리의 구조 및 공유 방법에 따라 분산메모리(distribute memory) 구조와 공유메모리(shared memory) 구조로 분류되는데, 멀티프로세서 시스템이 취하는 구조에 따라 시스템의 특성은 크게 달라진다[3]. 2004년을 기점으로 출시되었던 싱글 칩 기반의 멀티프로세서들은 대부분 공유메모리 멀티프로세서(shared memory multiprocessor)와 유사한 구조로 설계되었다[2].

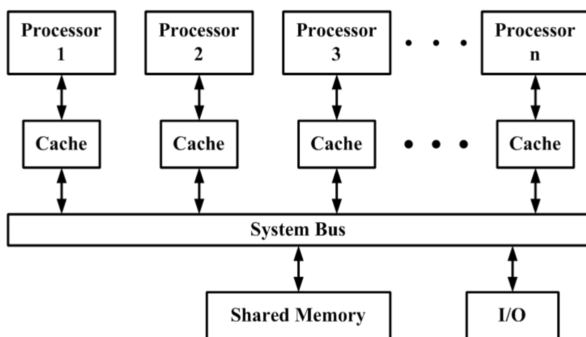

그림 1. 공유메모리 멀티프로세서 구조
Fig. 1. Architecture of a shared memory multiprocessor

공유메모리 멀티프로세서 시스템은 그림 1과 같이 하나의 공유메모리가 시스템 버스를 통하여 여러 프로세서에게 공유되어지는 구조로서, 각각의 프로세서들은 동시다발적인 공유메모리의 접근을 요구하기 때문에 프로세서들 간의 동기화는 항상 유지되어야 한다. 하지만 공유메모리 멀티프로세서 시스템에서 lock을 사용하는 전통적인 블로킹(blocking) 기반 프로세서 동기화 기법들은 멀티프로세서의 병렬성을 저해하고 다중처리 프로그래밍을 어렵게 할 뿐만 아니라 데드락(deadlock), 콘보잉(convoying), 우선순위 역전(priority inversion) 등과 같은 시스템에 치명적인 결과를 초래하는 문제점을 잠재적으로 가지고 있다[4]. 이러한 문제점들을 해결하기 위한 대안으로 트랜잭셔널 메모리가 제안되었다. 트랜잭셔널 메모리는 기존의 캐시 일관성 프로토콜을 일부 수정하여 구현된 논블로킹(non-blocking) 방식의 프로세서 동기화 방식으로서, lock을 사용하는 전통적인 동기화 방식의 문제점들을 모두 해결 할 수 있기 때문에 이를 실용화하기 위한 연구가 활발히 진행되고 있다[5]. 트랜잭셔널 메모리는 프로세서들 간의 경합 정도와 충돌 관리 정책에 따라서 트랜잭션 수행 속도 및 시스템 버스의 트래픽 정도가 크게 달라진다[6]. 특히 시스템 버스는 공유메모리 멀티프로세서 시스템의 성능과 규모를 결정짓는 매우 중요한 요소이며, 시스템 버스의 성능은 시스템 버스의 트래픽 관리 방법에 의해 크게 영향을 받는다[7].

이에 본 논문은 트랜잭셔널 메모리의 큰 분류 중 하나인 하드웨어 트랜잭셔널 메모리의 트랜잭션 동작을 충돌 관리 정책에 따라 네 가지로 구분하여 각각 모델링 하고, 프로세서들 간의 경합 정도에 따른 시스템의 성능과 시스템 버스의 트래픽 정도를 시뮬레이션을 통해 비교 분석한다. 그리고 분석된 시뮬레이션 결과를 통하여 시스템의 성능 향상에 가장 크게 기여할 수 있는 효율적인 충돌 관리 정책을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 트랜잭셔널 메모리의 개념과 트랜잭셔널 메모리의 충돌 관리 정책에 대해 설명하고 3장과 4장에서는 모델링 및 시뮬레이션 결과에 대해 설명한다. 마지막으로 5장에서는 본 논문의 결론을 맺는다.

Ⅱ. 관련 연구

2.1. 트랜잭셔널 메모리

Lock을 사용하는 블로킹 기반 프로세서 동기화 방식은 공유데이터에 접근하기 위한 선취권을 요구하는 방식으로서, 선취권을 획득한 프로세서에게만

임계영역에 접근하는 것을 허용하며 그렇지 못한 프로세서들은 블록(block) 처리한다. 블록 처리된 프로세서들은 선취권을 부여 받을 때까지 작업을 중단하고 대기해야 한다. 이러한 블로킹 방식의 프로세서 동기화는 멀티프로세서 시스템의 병렬적인 동작을 가로막고 순차적인 동작을 요구함으로써 시스템의 성능 저하를 야기한다. 트랜잭셔널 메모리는 lock을 사용하는 블로킹 기반 동기화 방식의 한계를 극복할 수 있는 프로세서 동기화 방식으로서, 여러 프로세서들이 공유데이터에 접근 할 때 충돌이 발생하지 않는 경우에 한해서 병렬적인 동작을 보장하는 논블로킹 기반의 기술이다[5].

트랜잭셔널 메모리의 동작은 트랜잭션(transaction) 단위로 수행된다. 트랜잭션이란 일련의 load/store 명령어들이 하나의 명령어와 같이 원자적(atomically)으로 수행되어지는 명령어 그룹이다. 트랜잭셔널 메모리는 공유데이터에 대한 선취권을 요구하지 않는 동기화 방식으로서, 프로세서들이 공유데이터에 접근할 때 우선 트랜잭션을 수행한 후 다른 프로세서들과의 충돌 여부를 검사한다. 충돌 검사에서 감지되는 충돌이 없다면 트랜잭션은 완료되지만, 만약 충돌이 감지된다면 수행했던 트랜잭션을 취소하고 공유데이터를 트랜잭션 수행 전의 값으로 복구한 후 다시 처음부터 트랜잭션을 시도한다. 여기서, 트랜잭션을 취소하고 데이터를 복구하는 동작을 abort라고 하고, 다른 프로세서와 충돌 없이 트랜잭션을 완료하는 것을 commit이라고 한다[5].

그림 2는 프로세서 P0, P1, P2가 공유영역의 공유데이터 A, B, C, D에 접근할 때, lock을 사용하는 블로킹 기반 방식과 트랜잭션널 메모리의 동작 과정을 비교 설명하는 그림이다. 트랜잭션의 경우, 프로세서들이 접근하는 공유데이터에 충돌이 발생하지 않으면 각 프로세서들은 공유영역에 대해 동시에 트랜잭션을 수행하여 commit에 도달한다. 하지만 P2와 같이 공유데이터 C의 충돌을 감지한 경우에는 수행했던 트랜잭션을 abort하고 다시 트랜잭션을 시도하여 commit에 성공한다. Lock의 경우에는 동일 공유데이터에 대한 동시 접근과 무관하게 lock을 소유한 프로세서에게만 배타적 영역의 진입이 허용되며 lock을 위해 대기하는 프로세서들의 작업은 순차적으로 처리된다. 즉, 동일한 환경에서 동일한 작업을 수행하더라도 중첩 가능한 작업을 병렬적으로 처리한 트랜잭션 메모리의 경우가 lock을 사용하는 순차적인 처리 방식인 블로킹 기반 방식보다 모든 작업이 완료되는 시점이 더욱 앞서는 것을 알 수 있다.

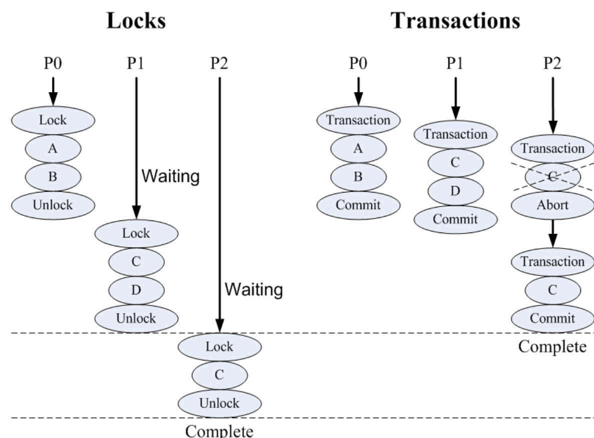

그림 2. Lock과 트랜잭션의 비교
Fig. 2. Comparison of locks and transactions

이러한 트랜잭셔널 메모리는 논블로킹 기반의 동기화 방식으로 프로세서들 간의 병렬적인 동작을 더욱 보장해 줄 뿐만 아니라 블로킹 기반의 기술들이 야기하는 데드락이나 콘보잉 등과 같은 위험을 배제할 수 있고, 프로세서들 간의 동기화를 트랜잭셔널 메모리에서 관리함으로 인해 병렬 프로그래밍을 더욱 쉽게 해주는 장점을 가진다.

트랜잭셔널 메모리는 트랜잭션을 구현하는 방법에 따라 소프트웨어 트랜잭셔널 메모리, 하드웨어 트랜잭셔널 메모리, 하이브리드(hybrid) 트랜잭셔널 메모리 등 크게 세 가지로 구분된다[8]. 소프트웨어로 구현된 소프트웨어 트랜잭셔널 메모리는 프로세서들 간의 다양한 경합 상황에 대한 유연한 대처는 가능하지만 모든 처리 방식이 소프트웨어에 의해 진행되므로 하드웨어 트랜잭셔널 메모리에 비해 오버헤드가 매우 큰 단점이 있다. 하드웨어 트랜잭셔널 메모리는 트랜잭션 수행 속도는 뛰어나지만 한정된 크기의 트랜잭션만 처리 가능하고 다양한 경합상황에 대처하지 못하는 단점이 있다. 소프트웨어와 하드웨어를 융합시킨 형태의 하이브리드 트랜잭셔널 메모리는 소프트웨어와 하드웨어의 장점을 모두 취하려 하였지만 실제 성능은 하드웨어 트랜잭셔널 메모리에 비해 매우 저조한 특성을 보였다. 하이브리드 트랜잭셔널 메모리의 성능을 기대했었던 수준으로 향상시키기 위해서는 소프트웨어에서 비롯되는 오버헤드를 크게 줄여야만 하는데 이는 쉽게 해결될 수 없는 문제이다. 그러므로 하이브리드 트랜잭셔널 메모리의 성능 향상을 위해서는 하드웨어

의 지원이 절실한 실정이며, 이에 따라서 하드웨어 트랜잭셔널 메모리의 다양한 경합 관리에 대한 연구는 더욱 중요시 되고 있다[8-10]. 이러한 이유로, 본 논문에서는 동일한 경합 상황에서 하드웨어 트랜잭셔널 메모리의 충돌 관리 정책에 따른 성능과 시스템 오버헤드를 비교 분석하기 위하여 하드웨어 트랜잭셔널 메모리를 모델링하여 연구를 진행하였다.

2.2. 트랜잭셔널 메모리의 충돌 관리 정책

동시에 수행되는 트랜잭션들이 commit에 성공하기 위해서는 트랜잭션 중인 공유데이터들 간의 충돌 여부를 관리해야 하며 충돌이 발생했을 경우 이미 수행했던 작업을 취소하고 데이터를 복구하는 등의 작업이 필요하다. 이와 같이 충돌이 감지되었을 때 수행되는 후처리 작업의 세부적인 동작은 충돌 관리 정책에 의해 결정되며 이는 트랜잭셔널 메모리의 특성을 결정짓는 가장 중요한 요소이다. 충돌 관리 정책은 충돌 감지(conflict detection) 정책과 데이터 관리(version management) 정책으로 구성되며, 각각은 동작 방식에 따라 eager 방식과 lazy 방식으로 분류된다[11].

충돌 감지 정책은 트랜잭션을 시도한 공유데이터의 일관성이 위배되었는지를 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 충돌 감지 방식은 충돌이 발생하는 즉시 감지가 가능하므로 프로세서들 간의 경합 정도가 높은 환경에서 효율적인 것으로 알려져 있다. 반면, lazy 충돌 감지 정책은 트랜잭션을 수행한 후 commit 직전에만 충돌 발생 여부를 검사하는 방식으로 충돌을 감지하는 시간은 eager 방식보다 느리지만 충돌 감지를 위한 오버헤드가 eager 방식보다 상대적으로 낮기 때문에 프로세서들 간의 경합이 치열하지 않은 환경에 효율적이다[11,12].

데이터 관리 정책은 쓰기 동작이 수반되는 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 수행 할 때 해당영역에 즉시 쓰기를 수행하고 원본데이터는 버퍼에 보존해 두는 방식이다. 이후 트랜잭션이 abort 되면 버퍼에 저장해둔 원본으로 공유데이터를 복구하고, 트랜잭션이 commit이면 추가적인 작업 없이 트랜잭션을 완료한다. Eager 데이터 관리 정책은 충돌 발생 시 버퍼와 공유영역의 데이터 이동에 대한 오버헤드가 발생하므로 프로세서들 간의 경합 정도가 낮은 시스템에 적합한 정책이다. Lazy 데이터 관리 정책은 트랜잭션 과정에서 쓰기 동작이 발생하더라도 해당영역에 직접 쓰기 동작을 수행하지 않고 버퍼에만 쓰기 동작을 수행한다. 이 후 트랜잭션을 commit 할 때 버퍼의 내용을 해당영역에 업데이트 한다. 만약 트랜잭션 과정에서 충돌이 발생한다면 수행했던 트랜잭션은 취소되지만 데이터 복구를 위한 추가적인 작업 없이 트랜잭션을 다시 시도한다. Lazy 데이터 관리 정책은 항상 일정한 수준의 오버헤드를 유지하므로 프로세서들 간의 경합이 높은 환경에 유용한 정책이다[11,12].

이러한 트랜잭셔널 메모리의 충돌 관리 정책들은 시스템 버스에 유발되는 오버헤드를 고려한 분석이 필요하다. 공유메모리 멀티프로세서 시스템은 프로세서들 간의 동기화, 캐시 일관성(cache coherence) 프로토콜, 공유메모리 접근 등이 모두 시스템 버스에 의해 동작하기 때문에 시스템 버스의 성능은 시스템의 성능과 직결된다. 트랜잭셔널 메모리 역시 공유메모리 멀티프로세서 환경에서 동작하므로 트랜잭션 동작은 시스템 버스를 기반으로 동작한다. 그러므로 트랜잭셔널 메모리 시스템을 설계 시에는 충돌 감지 정책 및 데이터 관리 정책에 따른 트랜잭셔널 메모리의 성능뿐만 아니라 트랜잭션 동작 중 유발되는 시스템 버스의 트래픽 역시 충분히 고려되어야 한다. 이에 본 논문은 충돌 관리 정책에 따른 트랜잭셔널 메모리의 성능과 시스템 버스의 트래픽 정도를 분석하기 위하여 하드웨어 트랜잭셔널 메모리를 충돌 관리 정책에 따라 각각 모델링하였다.

Ⅲ. 하드웨어 트랜잭셔널 메모리 모델링

하드웨어 트랜잭셔널 메모리의 충돌 관리 정책에 따른 성능과 시스템 오버헤드를 분석하기 위한 모델링은 C 언어를 사용하여 구현하였다. 모델링 된 하드웨어 트랜잭셔널 메모리의 구조는 그림 3과 같다.

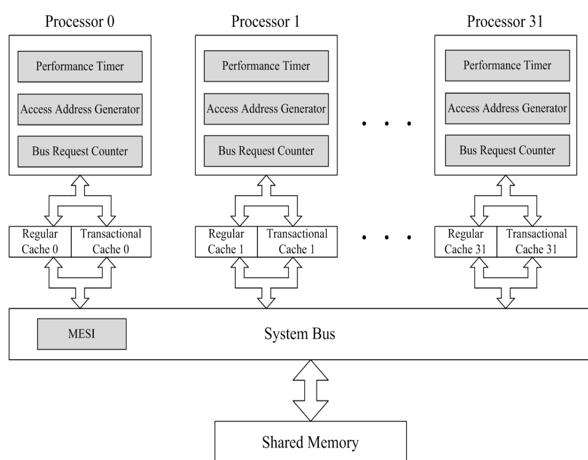

그림 3. 하드웨어 트랜잭셔널 메모리 시스템 모델의 구조
Fig. 3. Architecture of the modeled hardware transactional memory system

트랜잭셔널 메모리는 공유메모리 멀티프로세서 환경에서 동작하므로 모델링 역시 공유메모리 멀티프로세서 구조와 유사한 구조로 구현하였다. 공유메모리 멀티프로세서 구조는 시스템 버스의 성능에 따라 시스템의 규모가 결정되는데 현실적으로 수용 가능한 프로세서의 개수는 대략 30개 정도로 평가된다. 따라서 프로세서들 간의 경합 정도를 가장 가혹하게 설정하기 위해 32개의 프로세서가 시스템 버스를 통하여 하나의 공유메모리를 공유하는 시스템을 모델링 하였다[13]. 모든 프로세서에는 캐시메모리와 트랜잭셔널 캐시가 각각 할당되어 있어 공유메모리의 공유데이터를 참조하거나 트랜잭션을 수행하는데 사용되며 캐시 일관성 프로토콜은 MESI 프로토콜 방식으로 구현하였다. 캐시 쓰기 정책은 write-back 방식으로 구현 하였으며 쓰기 버퍼는 모델링의 간소화를 위해 생략하였다[13]. 프로세서 내부의 Performance Timer는 프로세서에 할당된 작업들을 모두 완료하는데 소비되는 시간을 측정한다. Access Address Generator는 프로세서가 할당된 작업을 수행하는 동안 접근해야하는 공유메모리의 주소를 생성하는 블록이다. 이 블록에서 생성되는 주소는 난수를 발생하여 생성하는데, 생성되는 주소의 개수를 프로세서의 개수보다 부족하게 설정함으로써 프로세서들의 경합 정도를 조절 하였다. Bus Request Counter는 프로세서가 트랜잭션을 수행 도중 시스템 버스의 사용을 요청했던 횟수를 측정하여 시스템 버스에 발생되는 트래픽 정도를 분석하는데 사용하였다.

이와 같은 모델링은 트랜잭셔널 메모리의 충돌 관리 정책에 따라 네 가지로 각각 구현하였다. 4가지 충돌 관리 정책은 충돌 감지 정책과 데이터 관리 정책에 따라 lazy_lazy, lazy_eager, eager_lazy, eager_eager로 구분되며, 각각의 모델링에 적용된 동작 방식은 그림 4의 순서도와 같다.

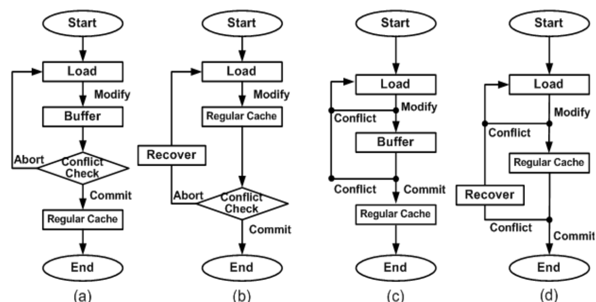

그림 4. 충돌 관리 정책에 따른 트랜잭션 동작 순서도: (a) lazy 충돌 감지, lazy 데이터 관리, (b) lazy 충돌 감지, eager 데이터 관리, (c) eager 충돌 감지, lazy 데이터 관리, (d) eager 충돌 감지, eager 데이터 관리
Fig. 4. Flowcharts of transaction operations according to the conflict management policy: (a) lazy conflict detection and lazy version management, (b) 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 충돌 감지 정책은 트랜잭션의 commit을 수행하기 직전에만 충돌 여부를 검사하고, 충돌이 감지되면 수행 중이던 트랜잭션을 무시하고 처음부터 다시 트랜잭션을 수행한다. 그림 4(c) 및 4(d)의 eager 충돌 감지 정책은 트랜잭션을 수행하는 과정에서 항상 충돌 여부를 감시하여 충돌이 감지되면 수행 중이던 트랜잭션을 무시하고 처음부터 다시 트랜잭션을 시도한다. 그림 4(a) 및 4(c) 의 lazy 데이터 관리 정책은 임의의 영역에 트랜잭션을 수행 한 후 트랜잭션을 commit 할 때만 레귤러 캐시의 해당 영역에 결과를 업데이트 한다. 그림 4(b) 및 4(d)의 eager 데이터 관리 정책은 트랜잭션 수행 과정에서 레귤러 캐시의 해당 영역에 작업 결과를 우선 작성하고, 만약 트랜잭션 결과가 abort이면 복사해둔 원본데이터를 이용하여 해당 영역의 데이터를 복구한다.

트랜잭션 수행 도중 발생하는 캐시메모리 및 공유메모리와 관련된 동작은 모델링의 간소화를 위하여 세부적인 구현은 생략하였으며 선행 조사한 연구들에서 사용된 시뮬레이터의 설정 값들을 참조하여 표 1 과 같이 지연 시간을 설정하고 적용하였다[14-16]. 단, 본 연구의 모델링 구현 환경에서 설정할 수 있는 최소 지연시간이 1 nsec 인 점을 감안하여

지연 시간이 가장 짧은 캐시메모리의 지연 시간을 기준으로 참조한 설정 값들을 동일한 비율로 조절하였다. 캐시 사상 방식은 직접 사상 방식으로 설정하였으며 버스 중재 방식은 in-order 방식으로 구현하였다. 충돌 감지 동작은 데이터 복구가 필요한 충돌과 데이터 복구가 필요하지 않은 충돌로 구분하여 측정하였으며, 충돌 검사를 위해서는 트랜잭션 캐시의 상태 비트를 읽기 위한 접근 시간이 요구된다. 데이터 백업을 위한 버퍼는 SRAM으로 가정하여 구현하였기 때문에 캐시메모리와 동일한 지연시간을 적용하였다. 이러한 설정들을 통한 전체 시스템 수행 시간은 Window API 에서 지원하는 QueryPerformanceCounter() 함수[17]를 사용하여 32개의 프로세서들이 동작을 시작하는 시점에서 32개의 프로세서들이 주어진 모든 동작을 완료하는 시점까지의 시간을 측정하였다.

표 1. 시뮬레이션에 사용된 시스템 파라미터
Table 1. System parameters used in simulations

Parameter	Settings
Number of Processor	32 processors
Regular Cache	SRAM, 1 ns access latency
Transaction Cache	SRAM, 1 ns access latency
Shard Memory	DDR2, 25 ns access latency
System bus	5 ns latency

Ⅳ. 실험 결과

앞장에서 설명한 모델링을 통하여 네 가지 트랜잭셔널 메모리 모델들 각각의 트랜잭션 수행 시간과 시스템 버스 요청 횟수를 측정하는 실험을 하였다. 트랜잭션 수행 시간은 32개의 프로세서들 각각이 64회의 트랜잭션을 모두 성공하는데 소비된 평균 시간으로서, 접근 가능한 공유데이터의 개수를 변화시키면서 경합 강도가 높은 경우에서 낮은 경우 모두를 대상으로 측정하였다. 트랜잭션 수행시간은 프로세서가 트랜잭션을 완료하는데 소모되는 시간만을 대상으로 하였으며 트랜잭션의 commit으로 인한 공유메모리 업데이트 동작은 측정시간에서 제외하였다. 시스템 버스 요청 횟수는 32개의 프로세서 각각이 64회의 트랜잭션을 모두 성공하기까지 시스템 버스를 요청한 평균 횟수를 측정한 것으로, 트랜잭션 수행시간 측정에서와 같이 접근 가능한 공유데이터 개수를 변화시키면서 측정하였다. 측정은 16회 반복하여 측정하였으며, 매회 마다 접근 가능한 공유데이터의 개수를 2의 배수로 증가하여 측정하였다. 프로세서들 간의 경합 정도가 난수에 의해 결정되는 점을 감안하여 더욱 객관성 있는 실험결과를 위해 동일한 실험을 반복하여 얻은 데이터의 평균값을 분석에 사용하였다.

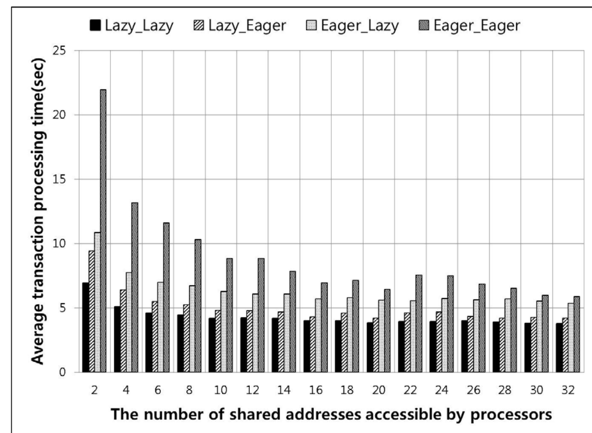

그림 5. 접근 가능한 공유데이터 개수에 따른 평균 트랜잭션 수행 시간
Fig. 5. Average transaction processing time according to the number of shared addresses accessible by processors

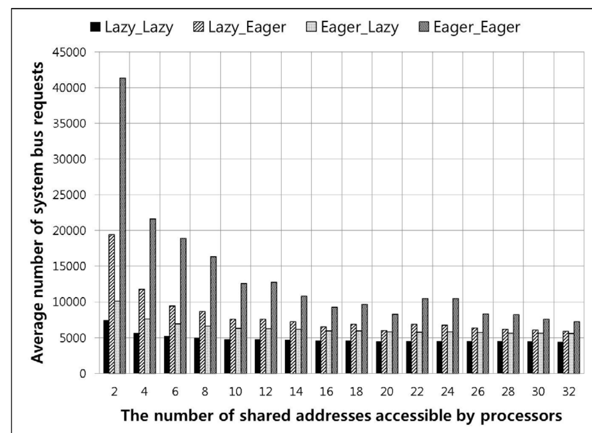

그림 6. 접근 가능한 공유데이터 개수에 따른 시스템 버스 평균 요청 횟수
Fig. 6. Average number of system bus requests according to the number of shared addresses accessible by processors

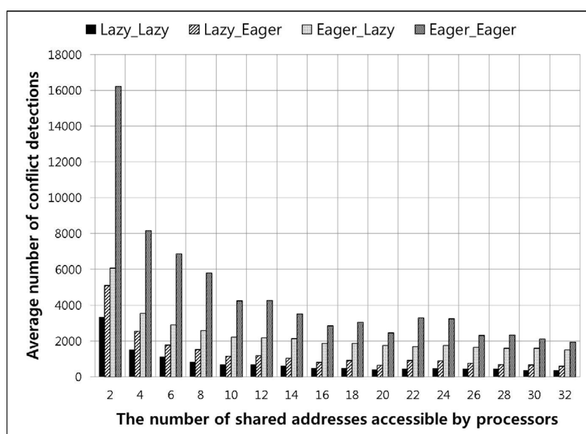

그림 7. 접근 가능한 공유데이터 개수에 따른 충돌 감지 평균 횟수
Fig. 7. Average number of conflict detections according to the number of shared addresses accessible by processors

실험 결과는 그림 5, 그림 6 및 그림 7 과 같다. 그림 5 는 32개의 프로세서들 각각이 64회의 트랜잭션을 수행하는데 소비된 시간을 측정한 결과로서 프로세서들의 경합 정도에 따라 각각의 충돌 관리 정책들에 대한 트랜잭션 전체 수행 시간을 나타내고 있다. 그림 6 은 32개의 프로세서들 각각이 64회의 트랜잭션을 완료하기 위해 시스템 버스를 요청했던 횟수를 측정한 결과이다. 그림 7 은 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 결과에서 프로세서의 경합 정도에 따른 영향을 가장 적게 받는 것으로 분석된다. 이 모델은 트랜잭션 수행 중 commit 직전에만 충돌 검사를 실행하므로 트랜잭션 과정의 불필요한 동작이 최소화되고 트랜잭션들의 병렬적인 충돌을 방지할 수 있으며[6,12], commit 시에만 공유영역에 트랜잭션 결과를 업데이트하기 때문에 트랜잭션 결과가 abort일 경우 원본 데이터를 복원하기 위한 시스템 버스 트래픽이 발생하지 않는다. 그러므로 트랜잭션 수행시간을 측정한 실험과 시스템 버스 요청 횟수를 측정한 실험에서 프로세서의 경합 정도 전반에 걸쳐 다른 모델들에 비해 가장 안정적인 성능을 보였다. 뿐만 아니라 이러한 충돌 감지 정책은 재시도 되는 트랜잭션 횟수를 최소화하기 때문에 트랜잭션 충돌 감지 횟수를 측정한 실험에서도 다른 모델들에 비해 낮은 수치를 기록하였다.

Lazy 충돌 감지 정책과 eager 데이터 관리 정책을 사용한 모델(lazy_eager)은 프로세서 간 경합이 높은 구간에서는 lazy_lazy 모델의 트랜잭션 수행 시간에 비해 측정된 수행 시간이 높게 측정되었으며, 경합이 낮은 구간에서는 lazy_lazy 모델의 트랜잭션 수행 시간과 비교적 가까운 값들이 측정되었다. 그리고 시스템 버스 요청 횟수의 경우 경합이 높은 구간에서 lazy_lazy 모델과 큰 차이를 보였으며, 트랜잭션 충돌 감지 횟수에서도 lazy_lazy 보다 더욱 많은 충돌을 감지한 것으로 나타난다. 이러한 결과는 eager 방식의 데이터 관리 정책이 공유데이터에 대한 업데이트 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.

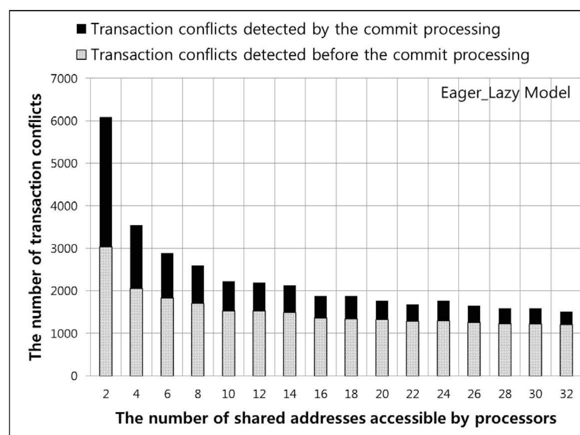

그림 8. Eager_Lazy 모델에서 검출되는 트랜잭션 충돌
Fig. 8. Transaction conflicts detected in the Eager_Lazy model

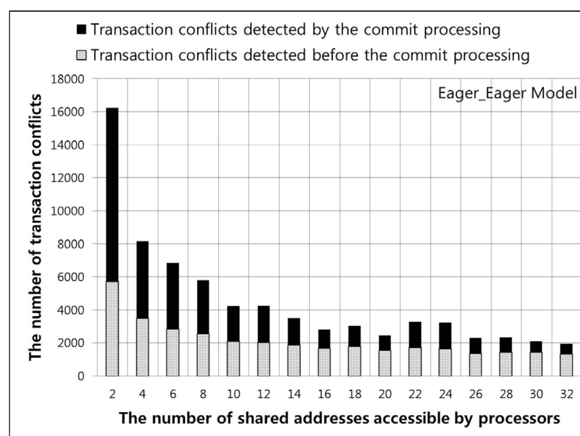

그림 9. Eager_Eager 모델에서 검출되는 트랜잭션 충돌
Fig. 9. Transaction conflicts detected in the Eager_Eager model

그림 8 과 그림 9 는 eager 충돌 감지 정책을 사용하는 모델들의 시뮬레이션 수행 중 감지된 충돌의 특성을 나타낸 그림이다. Eager 충돌 감지 정책을 사용하는 모델의 경우 commit 수행 과정에서 감지되는 충돌뿐만 아니라 commit 수행 이전의 단계인 트랜잭션 수행과정에서도 충돌이 감지될 수 있다.

Eager 충돌 감지 정책과 lazy 데이터 관리 정책을 사용한 모델(eager_lazy)은 lazy 충돌 감지 정책을 사용하는 모델들에 비해 트랜잭션 수행 시간이 높게 측정되었다. 트랜잭션 충돌 감지 횟수는 lazy_eager 모델의 트랜잭션 충돌 감지 횟수 보다 높게 측정되었지만 시스템 버스 요청 횟수는 lazy_eager 모델의 시스템 버스 요청 횟수 보다 낮게 측정되었다. 이러한 결과의 원인은, 충돌이 발생하더라도 즉시 충돌을 감지하지 못하고 commit 시점까지 트랜잭션을 진행한 후에 데이터 복구 작업을 하는 lazy_eager 모델과는 다르게, eager_lazy 모델은 그림 8 과 같이 트랜잭션 수행 과정에서도 충돌을 감지함으로써 불필요한 시스템 버스 요청이 줄어들고 데이터 복구로 인한 버스 트래픽이 없기 때문이다. 하지만 eager 방식의 충돌 감지로 인해 트랜잭션들의 병렬적인 충돌이 가중되고, 트랜잭션 과정에서 항상 충돌을 감지해야 하는 추가적인 수행시간이 요구된다.

Eager 충돌 감지 정책과 eager 데이터 관리 정책을 사용한 모델(eager_eager)은 프로세서들의 경합이 높은 구간에서 다른 모델들에 비해 트랜잭션 수행 시간과 시스템 버스 요청 횟수가 특히 높게 측정되었다. 이러한 결과는 eager 충돌 감지 정책으로 인해 트랜잭션 과정에서 충돌 감시에 수반되는 오버헤드가 증가하고 트랜잭션들의 병렬적인 충돌이 증가하기 때문이다. 또한 eager 데이터 관리 정책으로 인해 공유데이터의 업데이트 빈도가 증가하기 때문에 그림 9와 같이 전체적인 충돌 감지 횟수가 다른 모델들에 비해 증가하였고 데이터 복구가 요구되는 충돌 감지의 비중도 높아졌다.

본 장에서는 트랜잭셔널 메모리의 충돌 관리 정책에 따른 네 가지 모델들을 시뮬레이션 한 결과를 통해 각 모델들의 특징을 비교 분석하였다. 각 모델들의 성능과 특징을 분석해 본 결과, 프로세서들의 다양한 경합 상황에 대한 트랜잭션 수행 시간 및 시스템 버스 트래픽 유발 정도를 모두 고려하였을 때, lazy 충돌 감지 정책과 lazy 데이터 관리 정책을 사용한 모델이 최적화된 트랜잭션 동작을 통해 시스템의 성능 향상에 기여하는 정도가 가장 우수한 것으로 분석된다.

V. 결 론

본 논문에서는 트랜잭셔널 메모리의 충돌 관리 정책이 시스템 성능 및 시스템 버스 오버헤드에 미치는 영향을 알아보기 위하여, 충돌 관리 정책을 네 가지로 분류하고 각각에 대한 하드웨어 트랜잭셔널 메모리를 모델링 하고 시뮬레이션 하였다. 시뮬레이션 결과 충돌 감지 정책이 데이터 관리 정책에 비해 시스템의 성능에 더 큰 영향을 미치는 것으로 분석되며, 시스템 버스 오버헤드 측면에서는 데이터 관리 정책이 충돌 감지 정책에 비해 미치는 영향이 큰 것으로 분석되었다. 충돌 감지 정책의 경우 eager 방식보다 lazy 방식이 트랜잭션 수행 시간에서 더욱 좋은 성능을 보였으며, 데이터 관리 정책의 경우에서도 eager 방식보다 lazy 방식이 더욱 효율적인 것으로 나타났다. 충돌 관리 정책에 따른 네 가지 모델들의 시뮬레이션 결과, lazy 충돌 감지 정책과 lazy 데이터 관리 정책을 사용하는 것이 트랜잭션 수행 시간 및 시스템 버스 요청 횟수 측면에서 프로세서간의 경합 정도 전반에서 가장 효율적인 정책임을 알 수 있었다. 단, 이러한 결과는 각 프로세서들이 동일한 비율로 임의의 영역을 접근하여 동일한 작업을 수행한다는 가정에서 도출된 결론이기 때문에 제시된 결론의 일반화를 위해서는 더욱 다양한 변수들을 시뮬레이션에 적용하는 연구가 필요하다.

참 고 문 헌

[1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Uelick, The Landscape of Parallel Computing R*esearch: A View form Berkeley, Technical Report No. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences University of California at Berkeley*, 2006.

[2] J. Larus, R. Rajwar, *Transactional Memory*, Morgan & Claypool, 2010.

[3] J. May, *Parallel I/O for High Performance computing*, Morgan Kaufmann, 2001.

[4] T. Anderson, “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6-16, Jan. 1990.

[5] M. Herligy, J. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”, *Proceedings of ISCA 1993*, pp. 289-300, San Diego, CA, USA, May. 1993.

[6] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, D. Wood, “Performance Pathologies in Hardware Transactional Memory”, *Proceedings of ISCA 2007*, pp. 81-91, San Diego, CA, USA, Jun. 2007.

[7] J. Mellor-Crummey, M. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21-65, Feb. 1991.

[8] J. Larus, C. Kozyrakis, “Transactional Memory”, *Communications of the ACM*, vol. 51, no. 7, pp. 80-88, July. 2008.

[9] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, S. Chatterjee “Software Transactional Memory: Why is it Only a Research Toy?”, *Communications of the ACM*, vol. 51, no. 11, pp. 40-46, Nov. 2008.

[10] P. Damron, A. Fedorova, Y. Lev, “Hybrid Transactional Memory”, *Proceedings of ASPLOS-XII*, pp. 336-346, San Jose, CA, USA, Oct. 2006.

[11] K. Moore, J. Bobba, M. Moravan, M. Hill, D. Wood, “LogTM: Log-based Transactional Memory”, *Proceedings of HPCA-12*, pp. 254-265, Austin, Texas, USA, Feb. 2006.

[12] S.Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, M. Valero, “EazyHTM: Eager-Lazy Hardware Transactional Memory,” *Proceedings of MICRO-42*, pp. 145-155, New York, USA, Dec. 2009.

[13] D. Culler, J. Singh, A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.

[14] R. Yoo, H. Lee, “Adaptive Transaction Scheduling for Transactional Memory System”, *Proceedings of the SPAA 2008*, pp. 169-178, Munich, Germany, June. 2008.

[15] D. Burger, J. Goodman, A. Kägi, “Limited Bandwidth to Affect Processor Design”, *Micro, IEEE*, vol. 17, no. 6, pp. 55-62, Nov/Dec. 1997.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, “Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches”, *Proceedings of ISCA 2009*, Austin, pp. 184-195, TX, USA, June. 2009.

[17] Windows Desktop Development Center, http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904 (v=vs.85).aspx.

김 영 규 (Young-Kyu Kim)

2005년 2월 경주대학교 전자공학과 학사 졸업
2011년 2월 경북대학교 모바일통신공학과 석사 졸업
2011년 3월~현재 경북대학교 전자전기컴퓨터학부 박사과정
<관심분야> SoC, 컴퓨터 구조, 운영체제

문 병 인 (Byungin Moon)

1995년 2월 연세대학교 전자공학과 학사 졸업
1997년 2월 연세대학교 전자공학과 석사 졸업
2002년 2월 연세대학교 전기전자공학과 박사 졸업
2002년~2004년 하이닉스 반도체 선임연구원
2004년~2005년 연세대학교 연구교수
2005년~현재 경북대학교 IT대학 전자공학부 부교수
<관심분야> SoC, 디지털 VLSI, 컴퓨터 구조