논문 2011-48CI-1-10

집중 충돌 병렬 처리를 위한 효율적인 다중 코어 트랜잭셔널 메모리

(Multi-Core Transactional Memory for High Contention Parallel Processing)

김 승 훈*, 김 선 우*, 노 원 우**

(Seung Hun Kim, Sunwoo Kim, and Won Woo Ro)

요 약

다중 코어 프로세서의 보급과 더불어 이를 효율적으로 활용하기 위한 병렬 프로그래밍의 중요성은 나날이 강조되고 있다. 트랜잭셔널 메모리는 병렬 프로그래밍의 핵심적인 요소인 동기화(Synchronization)를 위해 제안된 구조로서 lock을 사용한 동기화로 인해 발생하는 병렬성 저하, deadlock 등의 문제를 극복할 수 있다. 본 논문은 높은 수준의 contention 상황에 따른 효율적인 트랜잭셔널 메모리의 구조에 대한 이론적인 분석을 제시하며 시뮬레이션을 통해 분석의 타당성을 확인한다. 시뮬레이션 환경은 하드웨어 트랜잭셔널 메모리(Hardware Transactional Memory) 시스템으로 구성되었으며 이론의 검증을 위해 STAMP 벤치마크와 높은 contention을 유발하는 프로그램을 시뮬레이션 하였다. 또한 트랜잭셔널 메모리를 적용한 dining philosopher problem의 모델링을 통해 효율적인 자원 할당 방안에 있어 lazy 데이터 관리 정책이 유리함을 보였다.

Abstract

The importance of parallel programming seriously emerges ever since the modern microprocessor architecture has been shifted to the multi-core system. Transactional Memory has been proposed to address synchronization which is usually implemented by using locks. However, the lock based synchronization method reduces the parallelism and has the possibility of causing deadlock. In this paper, we propose an efficient method to utilize transactional memory for the situation which has high contention. The proposed idea is based on the theoretical analysis and it is verified with simulation results. The simulation environment has been implemented using HTM(Hardware Transactional Memory) systems. We also propose a model of the dining philosopher problem to discuss the efficient resource management using the transactional memory technique.

Keywords : Transactional Memory, Multi-Core, Parallel processing, Synchronization, Multi-Threading

Ⅰ. 서 론

CPU 및 메모리의 작동주파수 향상은 컴퓨터의 전반적인 처리 능력을 증가시키며 이로 인해 알고리즘의 수정 또는 프로그램의 재개발 과정 없이도 소프트웨어 실행 시간을 비약적으로 단축할 수 있다. 그러나 2004년을 전후하여 CPU의 발전은 전력 소모 및 발열, clock skew 등의 이유로 한계에 부딪히게 되었다. 그 결과, 개별 프로세서의 성능을 높이는 것에 어려움을 겪게 된 인텔, AMD 등 주요 CPU 제조업체들은 여러 개의 프로세서를 하나의 칩에서 작동시키는 다중 코어 프로세서 개발에 집중하게 되었다.

이러한 다중 코어의 보급에 따라 소프트웨어 역시 달라진 하드웨어 환경을 활용해야만 효율적인 성능 향상

* 학생회원, ** 정회원, 연세대학교 전기전자공학과
(Dpartment of Electrical and Electronic Engineering, Yonsei University)
※ 이 논문은 2009년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2009-0077326)
접수일자: 2010년8월25일. 수정완료일: 2010년12월23일

이 가능하게 되었으며 필연적으로 병렬 프로그래밍의 중요성이 전에 없이 부각되었다. 병렬 프로그래밍에서 가장 핵심적인 요소라 할 수 있는 것은 서로 다른 코어에서 작동하는 쓰레드 간의 데이터 공유로 인한 충돌 문제 해결이다. 이를 효과적으로 해소하기 위한 기술을 동기화(Synchronization) 기법이라 하며 그림 1은 별개의 코어에서 공유 변수에 대해 실행되는 쓰레드 간의 동기화가 지켜지지 않아 문제가 발생하는 간단한 예를 보여준다.

현재 일반적으로 사용되고 있는 동기화 기법은 lock 방식이지만 이는 사용상의 오버헤드와 deadlock 또는 livelock 등의 문제를 유발할 수 있다. 최근에 개발된 트랜잭셔널 메모리는 lock 사용에 따른 이러한 단점을 개선하기 위해 제안된 것으로서 기본 개념은 데이터베이스 관리 시스템에 기초한다. 1993년 Herlihy 등이 최초로 하드웨어 지원을 바탕으로 작동하는 트랜잭셔널 메모리에 대해 발표[1]한 이래, 현재까지도 수많은 연구가 진행되고 있으며 다양한 형태의 트랜잭셔널 메모리에 대한 제안 및 성능 보고가 이루어지고 있다.

본 논문은 병렬 처리 프로그래밍에 있어서 높은 수준의 contention을 처리하는 트랜잭셔널 메모리의 효율적인 적용 방안을 제안하며, 세 가지 병렬 프로그램에서 트랜잭셔널 메모리를 활용한 모델링을 통해 효율적 자원 할당 방안을 논할 것이다. 일련의 과정은 full system simulator 인 Simics[2]와 다중 코어 환경에서 캐시 및 메모리 구조에 대한 시뮬레이션이 가능한 GEMS[3]를 사용하여 이루어졌다.

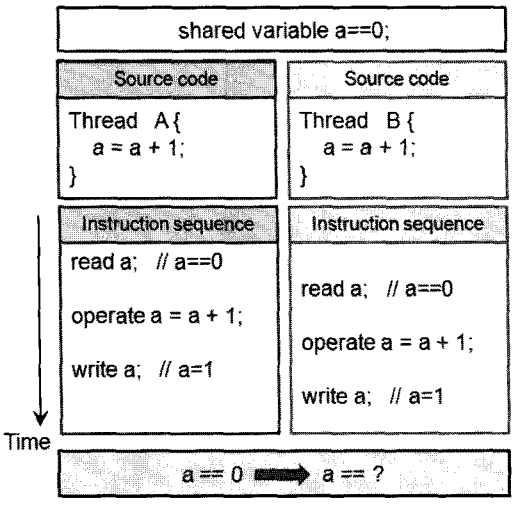


그림 1. 동기화 오류의 예
Fig. 1. The example of synchronization error.

본 논문의 구성은 다음과 같다. 먼저, Ⅱ장에서 트랜잭셔널 메모리와 lock의 실질적인 차이 및 특성과 이에 근거한 트랜잭셔널 메모리의 효율적인 활용방안에 대한 이론을 서술한다. Ⅲ장에서는 제안된 이론을 적용한 시뮬레이션을 통해 분석의 타당성을 입증하며 dining philosopher problem 모델링에 대한 결과를 서술한다. 마지막으로 Ⅳ장에서는 결론 및 향후 연구 과제에 대해 기술한다.

Ⅱ. 본 론

1. 트랜잭셔널 메모리와 lock의 차이

Lock 기반의 동기화 원리는 그림 2와 같이 공유 데이터에 대해 오직 하나의 쓰레드만 접근을 허용하는 크리티컬 섹션 (임계 구역) 설정에 있다.

이를 통해, lock을 획득한 쓰레드가 작업을 모두 마치고 이를 해제한 후에 다른 쓰레드가 작업할 수 있으므로 lock을 획득하지 못한 쓰레드는 작업을 중지하고 대기해야 한다. 이러한 배타적(exclusive) 특성으로 인해 결과적으로 다중 코어의 성능을 완전히 끌어내지 못하게 된다[4].

트랜잭셔널 메모리는 이와 같은 lock의 한계를 극복하는 것을 목적으로 한다. 트랜잭셔널 메모리의 실제는 단일 프로세스에 의해 실행되는 유한 순서를 갖는 명령어 집합(Instruction Set) 으로 정의할 수 있으며 트랜잭션이 이루어지는 순서는 다음 그림 3과 같이 나타낼 수

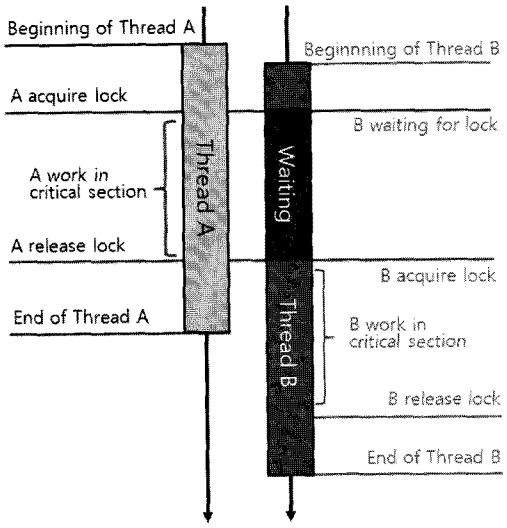


그림 2. Lock의 작동 원리
Fig. 2. Functional principle of lock.

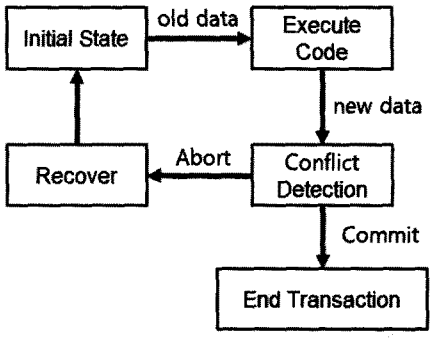


그림 3. 트랜잭션의 흐름도
Fig. 3. Flow of transaction.

있다.

이러한 구조의 트랜잭셔널 메모리는 여러 개의 불러오기/저장하기 명령어를 묶어서 하나의 명령처럼 실행(atomic execution)하며 lock과 달리 다중 코어의 공유 메모리 접근에 있어서 충돌(conflict)이 없을 것이라 가정하고 프로그램을 실행한다. 이후 충돌 발생을 감지한 경우에는 트랜잭션이 시작되기 이전의 상태로 되돌리며 이 때의 동작을 어보트(abort) 라고 지칭한다. 실제적인 충돌이 발생하지 않은 경우의 동작은 커밋(commit) 이라 하며 커밋 이후 트랜잭션은 종료된다. 그림 4, 5는 트랜잭셔널 메모리의 작동 원리를 충돌이 발생한 경우와 그렇지 않은 경우로 나누어 보여준다.

트랜잭셔널 메모리는 lock에 비해 여러 장점을 갖는다. 우선, 프로그램의 작성이 용이해지고 그에 따라 소프트웨어 개발 기간이 단축된다. Lock 사용 시 deadlock 및 livelock 방지를 위한 노력이 필요한 반면, 트랜잭셔널 메모리 사용 시 프로그래머는 원하는 구간에 트랜잭션을 선언해 주는 것만으로 공유 메모리 자원에 대한 동기화를 확보할 수 있기 때문이다. 또한 자원의 충돌이 없을 것이라는 가정 하에 코드 수행을 시작함으로써 크리티컬 섹션 설정에 의한 베타적인 특성이 없어지고 궁극적으로 다중 코어 프로세서의 하드웨어 활용을 극대화하는 효과가 있다.

2. 트랜잭셔널 메모리 적용 방안

트랜잭셔널 메모리 시스템은 소프트웨어 트랜잭셔널 메모리(Software Transactional Memory; STM)[5~7], 하드웨어 트랜잭셔널 메모리(Haraware Transactional Memory; HTM)[8~10], 하이브리드 트랜잭셔널 메모리(Hybrid Transactional Memory)[11~12]가 있다. 본 논문에서는 높은 contention이 발생하는 병렬 프로그램에 효율적이 HTM의 관리 정책 방법을 논의한다.

가. 데이터 관리 정책 (Version Management)

데이터 관리 정책은 발생 가능한 충돌에 대비하기 위해 원본 데이터의 보존 방법을 제공하며 그림 6 에서 보는 것과 같이 eager데이터 관리 정책과 lazy데이터 관리 정책으로 구분한다. Eager 데이터 관리 정책은 트랜잭션 중에 쓰기 동작이 발생한 경우 해당되는 메모리 주소에 바로 기록을 하고 원본 데이터를 다른 곳에 로

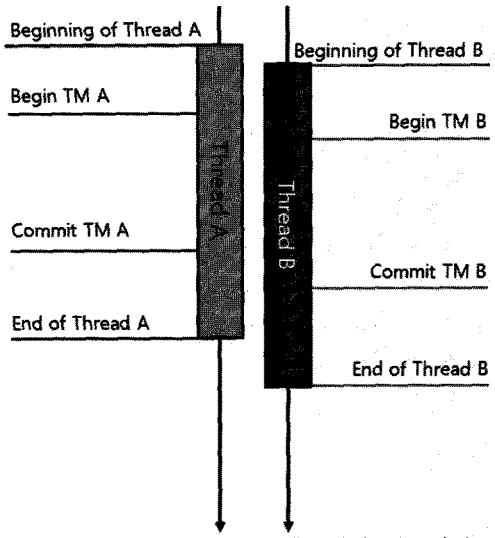


그림 4. 충돌이 없는 경우 트랜잭셔널 메모리의 작동원리
Fig. 4. Functional principle of transactional memory without conflict.

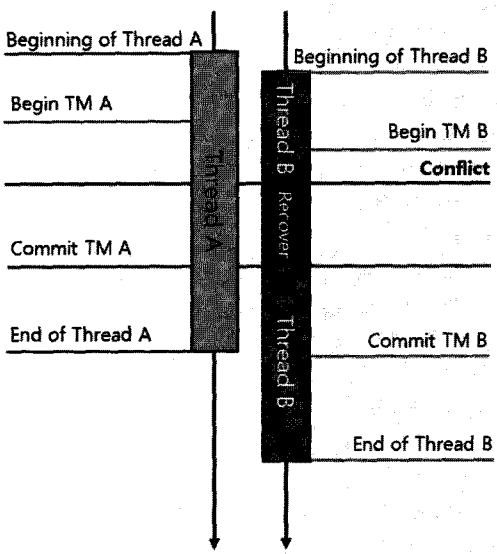


그림 5. 충돌이 있는 경우 트랜잭셔널 메모리의 작동원리
Fig. 5. Functional principle of transactional memory with conflict.

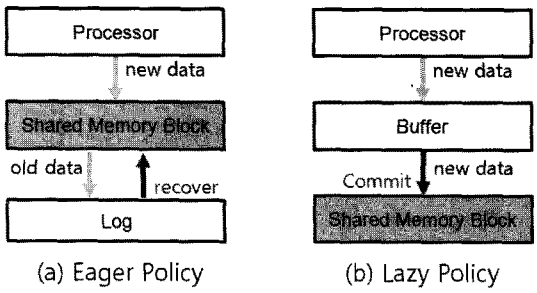


그림 6. 두 가지 데이터 관리 정책
Fig. 6. Two kinds of version management.

그로서 남겨 놓는 것이다. 반면 lazy 데이터 관리 정책은 메모리 주소에 바로 기록을 하지 않고 다른 곳에 기록해 놓았다가 트랜잭션이 성공적으로 종료될 경우 수정된 데이터를 모두 업데이트 한다.

데이터 관리 정책에 있어 오버헤드를 유발하는 작업은 메모리와 버퍼 사이의 데이터 이동 작업이다. Lazy 정책은 트랜잭션의 커밋 순간에만 데이터의 이동이 이루어지므로 어보트 발생 시의 오버헤드가 적다는 장점이 있다. 반대로 eager 정책의 경우, 새로운 데이터는 모두 메모리에 바로 기록되므로 커밋 순간의 오버헤드가 적어진다. 따라서 어보트가 빈번히 발생하는 트랜잭션의 수행에서 eager 정책을 사용할 경우, 각각의 어보트에 있어서 버퍼로부터 새로운 데이터를 메모리로 옮겨야 하므로 lazy 정책에 비해 비효율적이다.

동일한 트랜잭션 구간 내에서 충돌이 발생한 경우 오직 하나의 트랜잭션만이 커밋되어야 한다. 이는 커밋되지 않는 다른 모든 쓰레드는 어보트 되어야 함을 의미하며 따라서 동시 실행되는 쓰레드의 수가 늘어나면 충돌에 따른 어보트의 비율은 점차 증가하게 된다. 그러므로 트랜잭션 구간이 길어지고 쓰레드의 수가 늘어날수록 lazy 정책이 상대적으로 유리하게 사용될 수 있을 것이라는 결론을 도출할 수 있다. 이는 집중 충돌이 있는 병렬 처리 프로그램에 있어서 lazy 정책을 적용하는 것이 더욱 유리함을 의미하는 것이며 이에 대한 검증을 Ⅲ장의 시뮬레이션 결과를 통해 보이도록 한다.

나. 충돌 감지 정책(Conflict Detection)

충돌 감지 정책은 발생되는 충돌을 판단하여 잘못된 동작을 방지하기 위한 도구를 제공한다. 트랜잭션 간의 충돌은 프로그램의 수행 형태에 따라 어느 곳에서든 발생할 수 있으며 단일 트랜잭션 내에서도 충돌이 발생하는 시점은 상황에 따라 다르게 결정되므로 충돌 자체를

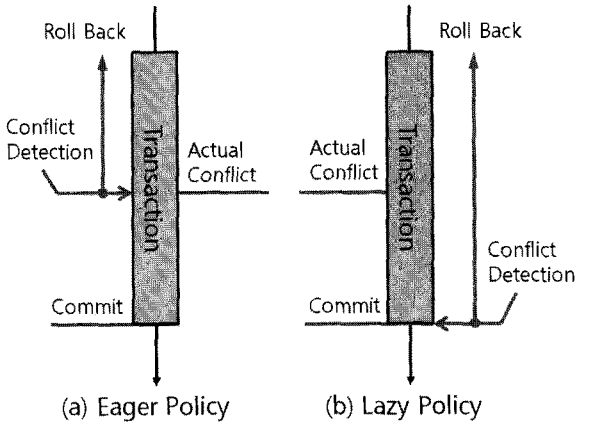


그림 7. 두 가지 충돌 감지 정책
Fig. 7. Two kinds of conflict detection.

예측한다는 것은 매우 어렵다. 따라서 모든 트랜잭션 수행 구간은 충돌이 발생하는지 감시되어야 하며 이를 처리하는 수단으로서 충돌 감지 정책이 필요하다. 그림 7은 충돌 감지 정책의 두 가지 방법인 eager 충돌 감지 정책과 lazy 충돌 감지 정책의 차이를 보여 준다.

Eager 정책을 사용할 경우 그 장점은 충돌이 자주 발생할 경우 나타난다. 충돌이 발생할 경우, 이를 해소하기 위해 트랜잭셔널 메모리는 시스템의 상태를 트랜잭션이 시작하기 전의 상태로 되돌리는 roll back 작업을 수행한다. Lazy 정책의 경우 커밋 시점까지 기다렸다가 충돌 여부를 판단하여 roll back을 수행하도록 한다. 따라서 충돌 시점부터 커밋 시점까지 무의미한 동작을 수행하게 되고 roll back 수행 시 그 처리량도 eager 정책에 비해 더 커지게 된다.

반면 lazy 정책은 충돌이 적은 상황에서는 유리할 수 있다. 트랜잭션 내의 어느 시점에서 발생할지 모를 충돌을 감지하기 위해 트랜잭션 내의 모든 동작에 대해 감지 작업을 해야 하는 부담을 커밋 이라는 한 시점으로 집중할 수 있기 때문이다. 또한 다수의 트랜잭션이 충돌을 일으키는 상황에 있어 연쇄적으로 얽힌 트랜잭션간의 충돌로 인한 성능 저하를 막을 수 있는 장점이 있다[13]. 하지만, 높은 수준의 충돌 상황을 나타내는 병렬 처리 프로그램의 경우, 기존의 eager 정책을 적용하는 것이 명백히 유리하다고 판단할 수 있다.

Ⅲ. 실 험 결 과

본 절에서는 시뮬레이션을 통한 벤치마크 수행 결과 및 앞장에서 제시된 효율적인 트랜잭셔널 메모리 적용

방안을 검증 한다. 트랜잭셔널 메모리 시스템은[14] GEMS를 이용하여 구현되었으며 시뮬레이션 환경은 Simics 기반의 16-Core 칩 멀티프로세서(CMP)로 구성되었다. 집중 충돌 병렬 처리 프로그램으로는 STAMP의 yada, Counter example, Dining Philosopher Problem을 사용하였다.

1. STAMP – yada

Yada 어플리케이션은 STAMP[15] 내의 어플리케이션 중에서 높은 수준의 contention을 보이는 것으로서 기본적으로 Delaunay mesh 개량을 위한 Ruppert 알고리즘을 구현한 것이다. 데이터 구조는 그래프이며, 여기에 모든 mesh triangle과 mesh 경계들의 집합, 개량되어야 할 triangle들을 담는 task queue 등이 저장된다. 대부분의 수행 시간이 triangle들의 재구성에 사용되며 상대적으로 긴 트랜잭션과 높은 contention이 발생한다.

본 연구에서 선정한 입력 데이터는 Spiral과 633.2 이다. 벤치마크의 수행은 다음과 같은 방식으로 진행되었다. 우선 lock을 사용한 형태의 yada 어플리케이션을 구현하여 수행한 결과를 기준 성능으로 삼고, 이를 트랜잭셔널 메모리 시스템에서 동작할 수 있는 형태로 수정하여 실행 결과를 비교한다. 이 때 두 가지 데이터 관리 정책, 즉 lazy 정책과 eager 정책을 각각 적용하도록 하며 충돌 관리 정책은 eager 정책을 적용한다. 이렇게 수행한 시뮬레이션의 결과는 그림 8과 같다.

시뮬레이션 결과에 따라 트랜잭셔널 메모리를 적용한 결과가 lock을 사용한 결과 보다 높은 성능을 보임을 확인할 수 있다. Contention이 높은 경우 공유 메모리 데이터는 쉽게 race condition에 놓이게 되며, lock을 사용할 경우 실제적인 충돌이 없더라도 크리티컬 섹션의 설정에 따라 무조건 순차적인 수행을 하게 된다. 반면 트랜잭셔널 메모리를 사용할 경우 contention이 높더라도 실제 충돌 발생 빈도에 따라 roll back의 횟수가 결정되므로 상대적으로 높은 성능을 보이는 것이다. 또한 높은 contention은 II장에서 서술한 것과 같이 어보트의 실행도 높이는 결과를 가져오기 때문에 충돌이 발생할 때 마다 실제 메모리의 값을 복원시켜야 하는 eager 정책 보다 커밋 단계에서 메모리 값을 업데이트 하는 lazy 정책이 더 높은 성능을 보임을 확인할 수 있다.

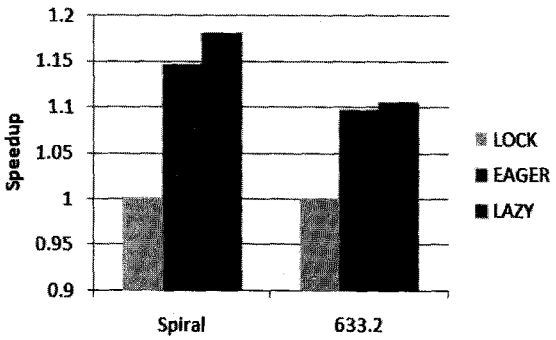


그림 8. Yada 시뮬레이션 결과
Fig. 8. Yada simulation result

2. Counter example

앞서 yada 어플리케이션을 통해 확인한 결과를 더욱 명확히 하기 위해 보다 간단한 형태의 시뮬레이션을 수행하였다. 이를 위해 공유 메모리에 위치한 카운터를 다수의 쓰레드가 각각 접근하여 정해진 횟수만큼 증가시키는 프로그램을 작성하였다. 프로그램은 각각 하나의 공유 카운터와 9개의 공유 카운터를 갖는 두 가지 형태이며, 공유 카운터의 수는 프로그램의 contention 정도를 의미한다. 각 쓰레드가 접근하여 race condition

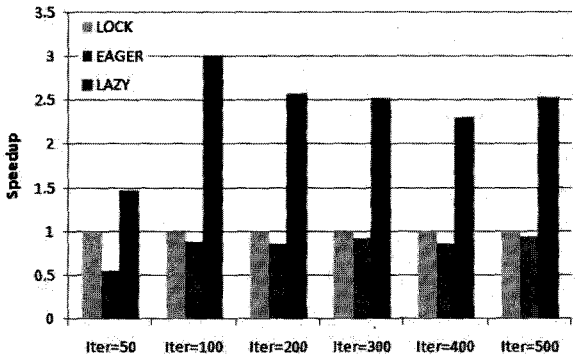


그림 9. 1개 카운터 프로그램 실행결과
Fig. 9. Single counter program simulation result.

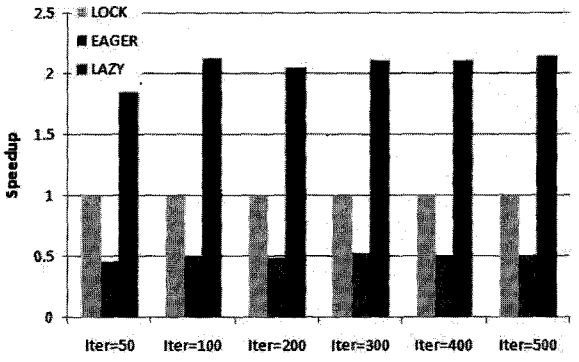


그림 10. 9개 카운터 프로그램 실행결과
Fig. 10. Nine counters program simulation result.

을 생성하는 카운터의 수가 증가할수록 contention이 높아진다고 볼 수 있기 때문에 9개의 카운터를 갖는 것은 1개의 카운터를 갖는 것보다 contention이 높은 프로그램이 된다. 그림 9와 그림 10은 각각 1개의 카운터와 9개의 카운터를 갖는 프로그램을 시뮬레이션 한 결과를 나타낸다.

Speedup을 통해 확인할 수 있듯이 lazy 데이터 관리 정책이 가장 빠른 수행시간을 갖는다. 이는 앞서 수행했던 yada 어플리케이션의 결과와 같은 것으로, 단순한 실험의 결과를 통해 그 효과를 더욱 명확하게 확인할 수 있다. 주목할 사항은 eager 데이터 관리 정책의 트랜잭셔널 메모리 적용이 lock을 사용하는 것보다 낮은 성능을 나타내는 것에 있다. 이는 공유 데이터의 변화를 즉시 메모리에 반영시키는 eager 데이터 관리 정책의 특성에 기인한 것으로서 contention 증가에 따라 roll back 페널티가 lock사용에 의한 프로그램 실행의 병렬성 저하 보다 커짐을 의미한다. 이러한 경향은 그림 9와 그림 10의 결과 비교를 통해 더욱 확실히 알 수 있다. 1개의 카운터만을 사용한 경우의 eager 데이터 관리 정책은 lock을 사용한 경우와 거의 비슷한 수준의 결과를 보이는 반면 9개 카운터가 사용될 경우 lock을 사용한 경우에 비해 절반 정도의 성능밖에 보이지 못한다. 즉, contention이 증가할수록 eager 데이터 관리 정책의 단점이 부각된다.

3. Dining Philosopher Problem

Dining Philosopher problem은 동기화와 deadlock problem을 설명하기 위해 널리 사용되는 예제로서 여러 프로세스가 동일한 자원을 요구할 때 deadlock에 빠지는 경우와 이를 해결하기 위한 방안을 설명하기 위해 주로 인용된다.

기존의 트랜잭셔널 메모리 연구범위는 응용프로그램 실행 시 크리티컬 섹션으로 정의된 구간을 두 개 이상의 프로세스가 동시에 접근하는 경우에 있어 최대한의 병렬성을 보장하는 것을 목적으로 한다. 따라서 연구 결과로 활용되는 벤치마크 프로그램은 특정 알고리즘을 병렬화 한 것이며 그 중 병렬화 할 수 없는 연산 또는 공유변수 접근에 대한 부분을 크리티컬 섹션으로 구분한 형태를 갖는다.

이는 응용 프로그램을 작성하고 활용하는 사용자 입장의 측면이 강조된 것이며 초기 제안된 트랜잭셔널 메모리의 주요 목적과도 일치한다. 그러나 dining philosopher problem과 같이 자원 할당 자체에 집중해야 하는 문제, 즉 크리티컬 섹션 내부 수행보다 크리티컬 섹션에 접근하는 것 자체가 중요한 경우 역시 트랜잭셔널 메모리를 통한 효율 향상을 기대할 수 있다. 이를 모델링하기 위해 표 1과 같은 프로그램을 작성하고 시뮬레이션을 진행하였다.

표 1. Dining philosopher problem 모델링에 사용된 코드의 주요 부분

Table 1. Main part of dining philosopher problem modeling source code.

```
while(ate<global_info.numberofeat){
  BEGIN_WORKLOAD_TRANSACTION

    BEGIN_TRANSACTION(0)
    //Start of critical section

      forkauth[myid]++;
      forkauth[(myid+1)%num_ph]++;

      if ( (forkauth[myid]==1) &&
     forkauth[(myid+1)%num_ph]==1) )
        ate++;

      forkauth[myid] = 0;
      forkauth[(myid+1)%num_ph] = 0;

    COMMIT_TRANSACTION(0)
    //End of critical section

  END_WORKLOAD_TRANSACTION
}
```

하드웨어 트랜잭셔널 메모리 시스템이 지원되지 않는 경우에 프로그램은 잘못된 자원할당을 복구할 수 있는 능력이 없다. 따라서 크리티컬 섹션의 시작점에서 오직 하나의 프로세스만이 진입할 수 있으며 이것이 병렬성을 저하시키는 요인이 된다. 동일한 경우에 있어 트랜잭셔널 메모리는 다중 접근을 허가하며 이는

```
forkauth[myid]++;
forkauth[(myid+1)%num_ph]++;
```

로 표현된다. 이렇게 자원 할당이 이루어진 상태에서 문제가 발생하지 않음을 아래 코드로 판단한다. 그 결과 목

적 연산인 ate++ 를 실행한다.

```
if ( (forkauth[myid]==1) &&
(forkauth[(myid+1)%num_ph]==1) )
```

앞서 언급한 것과 같이 dining philosopher problem 역시 특정 구간 내에서 각 프로세서들이 크리티컬 섹션 내부로의 진입을 지속적으로 요구하는 높은 contention 의 문제로 간주할 수 있다. 따라서 이를 해결하기 위해서는 lazy 데이터 관리 정책의 사용이 적합할 것이라 예상할 수 있다. 뿐만 아니라 자원 할당 가능성의 여부에 따라 실제 목적 연산을 수행하는 것이 뒤로 미루어지기 때문에 변경된 데이터의 값을 미리 공유 캐시에 저장하는 eager 데이터 관리 정책은 프로그램 진행에 유용하지 못할 것으로 예측할 수 있다. 그림 11에 나타난 시뮬레이션 결과는 이러한 예측이 일치하고 있음을 보여준다.

각 시뮬레이션은 iteration 횟수로 구분되며 이는 transaction의 길이로 정의할 수 있다. Speedup은 lock을 사용한 경우를 기준으로 계산한 것이며 시뮬레이션 상의 모든 경우에 있어 lazy 데이터 관리 정책이 더욱 뛰어난 성능을 보임을 확인할 수 있다.

가장 적은 100회 iteration의 경우에 있어 eager 데이터 관리 정책은 lock을 사용한 방법과 크게 다르지 않은 성능을 보인다. 이는 트랜잭션의 취소에 따른 roll back의 단점이 lock을 사용해 오직 하나의 프로세스에만 자원을 할당하는 것과 상충될 정도로 크기 때문이다. 반면에 lazy 데이터 관리 정책은 시뮬레이션 전반에 걸쳐 최고 60%, 평균적으로 40%이상의 성능 향상을 보이고 있다. 트랜잭션의 길이가 짧을수록 eager 데이터 관리 정책에 따른 단점이 부각되며 이는 트랜잭션의 길이가 길어질수록 lazy 정책과 eager 정책간의 성능 차이가 줄어드는 것을 통해서도 확인할 수 있다.

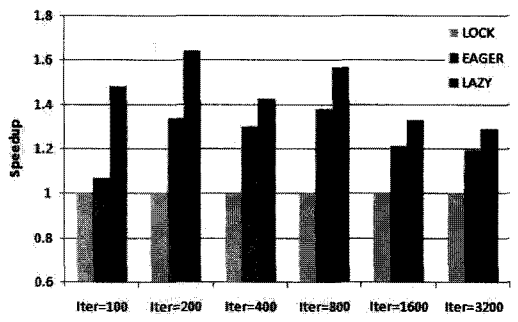


그림 11. Dining philosopher problem의 시뮬레이션 결과
Fig. 11. Dining philosopher problem simulation result.

Ⅳ. 결 론

본 연구를 통해 다중 코어 상에서의 병렬 처리 프로그램의 개발 시, contention이 높은 어플리케이션의 수행에 있어서는 lazy 데이터 관리 정책이 더 유리함을 볼 수 있었다. 세 개의 벤치 마크 프로그램을 통한실험 결과를 기반으로 하여 contention 정도에 따른 데이터 관리 정책의 영향을 좀 더 명확하게 파악할 수 있었다.

연구 결과를 통해 알 수 있듯이 트랜잭셔널 메모리는 적용되는 병렬 프로그램의 충돌 정도에 따라 다른 성능이 관찰된다. 따라서 추후 데이터 충돌 감지 정책에 대한 비교 분석을 통한 효율적인 적용 방안에 관한 연구를 진행할 예정이다.

참 고 문 헌

[1] Maurice Herlihy and J. Eliot B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures, in *Proceedings of the 20th Annual International Symposium on Computer Architecture*. pp. 289--300, 1993.

[2] Peter S. Magnusson Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform, *IEEE Computer*, 35(2):50-58, Feb. 2002.

[3] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92-99, Sept. 2005.

[4] Peng Wu and et al. in IBM Research, Compiler and Runtime Techniques for Software Transactional Memory Optimization, *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*. 21 (1):7-23, Jan 2009.

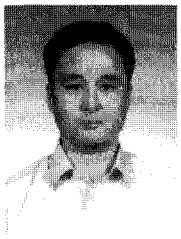
[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. *Lecture Notes in Computer Science*, Volume 4167, 194-208, 2006.

[6] K. Fraser and T. Harris. Concurrent Programming Without Locks. *ACM Trans. on Computer Systems*, 25(2), May 2007.
[7] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRTSTM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. 11th *ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.
[8] L. Hammond, V.Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H.Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency, 31st *Intl. Symp. on Computer Architecture*, June 2004.
[9] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. 32nd *Intl. Symp. on Computer Architecture*, June 2005.
[10] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proc. of the Eleventh IEEE Symp. on High-Performance Computer Architecture*, Feb. 2005.
[11] C. C. Minh, M. Trautmann, J. Chung,, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. 34th *Intl. Symp. on Computer Architecture*, June 2007.
[12] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. 34th *Intl. Symp. on Computer Architecture, June 2007. TR 910, Dept. of Computer Science*, Univ. of Rochester, Dec. 2006.
[13] Tomic, Sasa and Perfumo, Cristian and Kulkarni, Chinmay and Armejach, Adria and Cristal, Adrian and Unsal, Osman and Harris, Tim and Valero, Mateo, EazyHTM: Eager-Lazy Hardware Transactional Memory, MICRO 09 *Proceedings of the 2009 42nd IEEE/ACM International Symposium on Microarchitecture*, 2009.
[14] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in Procs. of the 13th *Intl Symp on High-Performance Computer Architecture*, pages 261-272, Feb. 2007.
[15] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in Proceedings of the *IEEE International Symposium on Workload Characterization*, pages 35-46, 2008.

저 자 소 개

김 승 훈(학생회원)
2009년 연세대학교 전기전자 공학과 졸업 (학사)
2009년 연세대학교 전기전자 공학과 입학 (석사과정)
<주관심분야 : 컴퓨터 시스템, 트랜잭셔널 메모리 등>

김 선 우(학생회원)
2009년 연세대학교 전기전자 공학과 졸업 (학사)
2009년 연세대학교 전기전자 공학과 입학 (석사과정)
<주관심분야 : 임베디드 시스템, 트랜잭셔널 메모리 등>



노 원 우(정회원)
1996년 연세대학교 전기공학과 졸업 (학사)
1999년 University of Southern California 졸업 (석사)
2004년 University of Southern California 졸업 (공학박사)
2003년~2004년 Apple Computer Inc. 인턴 연구원
2004년~2007년 California State University 전기 및 컴퓨터공학과 조교수
2006년~2007년 ARM Inc. 소프트웨어 엔지니어
2007년~현재 연세대학교 전기전자공학과 조교수
<주관심분야 : 고성능 마이크로프로세서 디자인, 컴파일러 최적화, 임베디드 시스템 디자인 등>