트랜잭셔널 메모리 시스템의 성능향상을 위한 선택적 트랜잭셔널 메모리 충돌해결정책

전원, 노원우
연세대학교 전기전자공학과
e-mail : jeonwon@yonsei.ac.kr, wro@yonsei.ac.kr

Selective Conflict Resolution for Transactional Memory System to Improve Performance

Won Jeon, Won Woo Ro
School of Electrical and Electronic Engineering, Yonsei University

요 약

트랜잭셔널 메모리는 다중 코어 시스템에서 lock 을 대체할 메모리 동기화 기법으로 소개되었다. 트랜잭셔널 메모리를 사용하는 시스템에서 같은 주소의 메모리에 동시에 접근하여 충돌이 일어난 트랜잭션은 충돌해결정책에 의해 유효화 될지 버려질지 선택된다. 기존의 트랜잭셔널 메모리는 고정된 충돌해결정책을 사용하여, 상황에 따라 가장 유리한 트랜잭션을 선택 해주지 못하는 한계가 있었다. 본 논문에서는 상황에 따라 여러 정책 중 유리한 충돌해결정책을 판단하여 적용시키는 방법을 제안한다. STAMP 벤치마크를 통한 시뮬레이션 결과, 제안하는 방법은 기존에 사용되는 Timestamp, Karma 충돌해결정책 대비 평균 22% 높은 성능 향상을 보였다.

1. 서론

트랜잭셔널 메모리는 다중 코어 시스템에서 lock 을 기반으로 하는 공유 변수 동기화의 문제를 해결하기 위해 제안되었다. 기존 lock 이 갖는 과도한 동기화에 의한 성능 저하, deadlock 의 발생 등의 문제를 없애고, 병렬 프로그래밍의 난이도를 완화하여 다중 코어 시스템의 자원을 효율적으로 사용하는 것이 트랜잭셔널 메모리의 목표이다. 트랜잭셔널 메모리는 다중 코어 동기화 기법의 가장 유망한 방법으로 여겨져, 활발한 연구가 진행 중이며 최신의 프로세서에 구현이 되기도 하였다 [1][2].

트랜잭셔널 메모리를 사용하는 시스템에서 같은 주소의 메모리에 동시에 여러 스레드가 쓰기를 시도하거나, 한 스레드가 값을 읽은 주소에 다른 스레드가 쓰기를 시도할 때, 충돌 (Conflict) 이 발생했다고 정의한다. 트랜잭셔널 메모리에는 충돌이 발생하였을 때, 여러 스레드 중 하나의 스레드만을 선택하여 유효화 해주고 다른 스레드의 일은 모두 무효화 시킨다. 이를 충돌해결정책 (Conflict Resolution) 이라 한다.

기존 트랜잭셔널 메모리의 충돌해결정책은 한 가지 방법으로 고정되어 사용되었다. 하지만 메모리 충돌을 일으킨 트랜잭션들의 특성과, 충돌이 발생한 주변 상황에 따라 유리한 해결방안이 달라질 수 있다 [8]. 따라서 고정된 충돌해결정책으로는 특정 상황에서 가장 좋은 성능을 얻을 수 있는 트랜잭션을 선택하지 못하는 문제가 있었다.

본 논문은 트랜잭셔널 메모리 시스템에서 충돌이 발생하였을 때, 가장 손해가 적은 트랜잭션을 버리도록 선택하는 Selective Conflict Resolution (SCR) 를 제안한다. SCR 은 상황에 따라 유동적으로 정책을 바꾸어, 주어진 상황에서 성능 이득을 가장 크게 얻을 수 있는 트랜잭션을 선택한다. 결과적으로 기존의 고정된 충돌해결정책을 사용한 시스템 대비 평균 22% 증가된 성능을 보였다.

본 논문의 구성은 다음과 같다. 2 절에서는 트랜잭셔널 메모리에 대한 전반적인 내용과 기존 정책의 특징 및 문제점을 설명하고, 3 절에서는 이 문제점을 해결할 제안하는 충돌해결정책을 설명한다. 4 절에서는 시뮬레이션을 통해 성능을 검증한다. 마지막으로 5 절에서는 결론 및 향후 연구 방향에 대해 서술한다.

2. 관련 연구

2.1. 트랜잭셔널 메모리

다중 코어 시스템에서 공유 변수 동기화를 위해 기존에 사용되고 있는 lock 방식은 공유 자원으로 사용되는 변수를 독점하여, 한 번에 오직 한 스레드 만이 접근할 수 있도록 허락한다. Lock 을 선점하지 못한 스레드는 상대방의 lock 이 풀릴 때까지 다른 일을 하지 못하고 기다리게 된다. 이러한 특성 때문에 다중 코어를 효과적으로 사용하지 못해 성능 저하가 발생한다.

트랜잭셔널 메모리는 M. Herlihy 에 의해 공유자원을 사용하는 다중 코어 컴퓨터 메모리에 적용되었다 [1]. 트랜잭션은 직렬성과 원자성을 갖는 명령어의 집

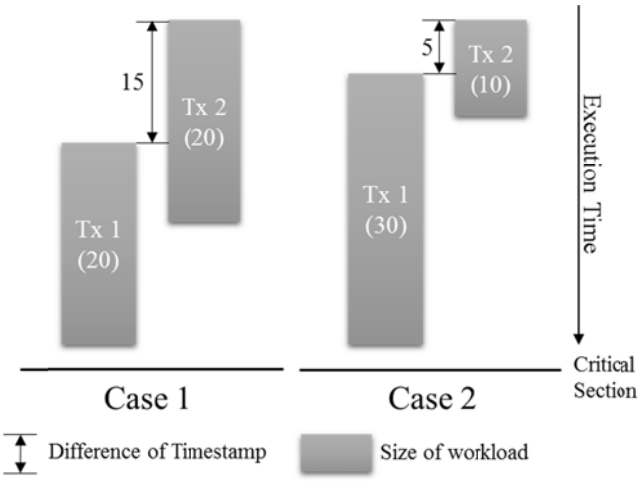
합이다. 즉, 더 이상 분리되거나, 다른 트랜잭션이 한 트랜잭션의 중간에 끼어들지 못하며, 실행이 끝나면 해당 트랜잭션에서 변한 값이 모두 적용되거나 또는 모두 버려지게 되는 성질을 가진다. 트랜잭셔널 메모리는 이 트랜잭션을 기본 단위로 가진다. 트랜잭셔널 메모리가 사용된 시스템에서는 여러 스레드들이 같은 시간에 공유자원에 접근을 하게 프로그래밍 되어 있어도, 모든 스레드가 자유롭게 변수에 접근이 가능하여 각 스레드가 멈추지 않고 진행된다. 그 후 트랜잭션이 끝날 때 사용된 공유자원이 다른 스레드에 의한 변형이 있었는지 확인한다. 확인 결과, 충돌이 없었다면, 해당 트랜잭션이 유효화 되어 메모리에 적용된다. 만일 충돌이 확인된다면, 정해진 규칙에 의해 충돌이 발생한 여러 트랜잭션 중 우선순위를 갖는 하나의 트랜잭션만을 골라 유효하게 해준다. 나머지 트랜잭션에 의한 변화는 모두 버려지고 해당 트랜잭션들이 시작되기 이전 상태로 되돌아 간다. 이 때 트랜잭션의 결과가 유효화 되어 메모리에 적용 되는 것을 커밋(commit) 이라 하고, 변화가 모두 버려지는 것을 어보트 (abort) 라고 지칭한다. 또한 트랜잭션이 어보트 되어 트랜잭션 실행 이전 단계로 돌아가는 것을 롤백(roll-back)이라고 한다.

트랜잭셔널 메모리가 기존의 lock 과 다른 부분은 공유 자원을 독점하지 않는다는 것이다. 선점한 스레드 외에는 모든 접근이 불가능한 lock 과 달리 트랜잭셔널 메모리는 접근을 허락하고, 충돌이 발생한 후에 문제를 해결한다. 이 차이점은 트랜잭셔널 메모리로 하여금, lock 이 갖는 deadlock 등의 문제를 해결하게 만들어, 다중 코어 시스템을 더 쉽게, 더 많이 활용할 수 있도록 해준다 [1][3].

2.2. 충돌해결정책

충돌해결정책은 트랜잭션 간에 충돌이 발생하였을 때, 어떤 트랜잭션을 어보트 할지 결정하는 역할을 한다. 충돌해결정책은 어보트를 시켜도 손해가 가장 적은 스레드를 골라서, 롤백 시 버려지는 자원을 최소화할 수 있다. 충돌이 발생하여 트랜잭션이 어보트 되면, 해당 트랜잭션에서 완료된 연산이 모두 취소되고 트랜잭션의 처음부터 다시 시작되어야 한다. 이는 곧 전력, 시간 등의 자원 낭비로 이어지고 성능 저하를 유발한다. 따라서 충돌이 발생하였을 때, 어보트 되어도 손해가 가장 적은 트랜잭션을 고르는 충돌해결정책의 역할이 성능에 큰 영향을 미친다 [7][8][9]. 이처럼 충돌해결정책이 트랜잭셔널 메모리의 성능에 중요한 이슈가 되기 때문에, 여러 정책들이 제시되었다. 아래는 대표적인 충돌해결정책의 예시이다.

- Suicide: 공유변수 충돌을 먼저 발견한 트랜잭션을 스스로 어보트 시킨다. 오버헤드가 가장 적은 충돌해결정책 중 하나로, 성능보다 적은 오버헤드를 우선시 할 때 사용된다.
- Aggressive: Suicide 와 반대로, 충돌을 발견한 트랜잭션이 상대방을 어보트 시킨다. Suicide 와 마찬가지로 오버헤드가 적다. 하지만 Suicide 와 Aggressive 는 롤백 시 낭비되는 자원과 주어진 상황을 고려하지 않고 어보트할 트랜잭션을 선택하기 때문에 좋은 성능을 기대하기는 어렵다.
- Backoff: 먼저 충돌을 발견한 트랜잭션이 어보트 되고, 해당 트랜잭션이 일정 시간 동안 재진입 하지 못하게 한다. 어보트 된 트랜잭션이 바로 재진입 할 시, 이전에 충돌을 일으킨 트랜잭션과 재 충돌을 일으켜 다시 어보트 될 확률이 매우 높다.
- Timestamp: 충돌을 일으킨 트랜잭션 중, 먼저 생성되었던 트랜잭션을 커밋 한다. 오랫동안 실행된 트랜잭션은 롤백 시 낭비되는 자원이 많기 때문이다. 이를 구현하기 위해, 각 트랜잭션이 시작될 때마다 시간을 기록하여 저장해야 한다. 충돌해결정책이 불려질 때 마다, 시간을 비교하여 어보트 될 트랜잭션을 선택해야 한다. 위의 세 충돌해결정책과 달리 오버헤드가 큰 반면 성능에서 더 이점을 가질 수 있다.
- Karma: 작업량이 가장 많은 트랜잭션을 커밋 해준다. 작업량이 많아 긴 실행시간이 필요한 트랜잭션은 롤백 시 버려지는 자원이 크기 때문이다. 작업량의 기준으로는 R/W 변수의 총 개수를 주로 사용한다. 이를 구현하기 위해 각 트랜잭션들의 R/W 개수를 기록해야 한다. 즉 Timestamp 와 같이 오버헤드가 큰 반면, 더 이점이 되는 트랜잭션을 선택할 수 있다.



(그림 1) 두 스레드의 트랜잭션이 충돌한 상황

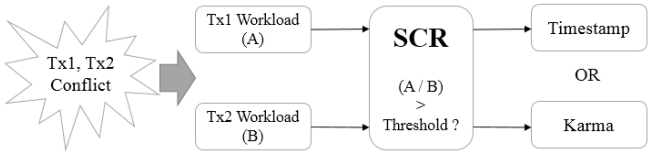
Timestamp 와 Karma 충돌해결정책 중 한 가지로 고정을 하여 사용한다면 그림 1 과 같은 상황에서 불이익을 볼 수 있다. 그림 1 의 Case 1 과 같은 상황에서 Timestamp 를 적용시킨다면, Tx2 를 커밋 해서 Tx2 와 Tx1 의 시작 시간 차이인 15 만큼의 이득을 볼 수 있다. 하지만 Karma 를 적용시킨다면, Tx1 과 Tx2 의 작업량이 같기 때문에, Suicide 와 같은 작동을 하게 되어 Tx2 가 어보트 될 상황이 발생할 수 있다. 반면, Case 2 에서는 비록 Tx2 가 먼저 시작 되었지만, Timestamp 를 적용시켜서 5 의 이득을 보는 것 보다, Tx1 을 선택하여 두 트랜잭션의 작업량 차이인 20 만

큼의 이득을 보는 것이 유리하다. 즉, Case 1 에서는 Timestamp, Case 2 에서는 Karma 를 적용시킬 때, 가장 좋은 성능을 보일 수 있다. 기존의 충돌해결정책은 한 가지 정책으로 고정되기 때문에, 이와 같은 상황에서 적절한 대응을 하지 못한다.

3. 제안하는 방법

제안하는 방법은, 공유자원 충돌을 일으킨 두 트랜잭션을 간단한 수식으로 비교하여, Timestamp 와 Karma 정책을 유동적으로 사용하는 SCR 이다. 2.2 절에서 설명한 기존 충돌해결정책의 문제점을 통해 알 수 있듯이 두 트랜잭션의 작업량 차이가 클 때는 Karma 를 적용시키는 것이 유리하고, 그렇지 않을 때는 Timestamp 를 적용시키는 것이 유리하다. 또한 충돌해결정책의 오버헤드를 최소한으로 사용하여 유리한 트랜잭션을 선택할 수 있어야 한다.

SCR 은 위의 상황을 고려하여, 충돌한 두 트랜잭션 중 하나의 작업량을 다른 하나의 작업량으로 나누어, 그 값이 정해진 임계 값 이상으로 차이가 나면 Karma 를 사용하고 이하일 때는 Timestamp 를 사용한다. SCR 에서는 이 임계 값이 성능에 영향을 미칠 수 있다. 그림 2 는 SCR 의 전반적인 구조를 나타낸다.



(그림 2) SCR 의 전반적인 구조

4. 실험

4.1. 실험 환경

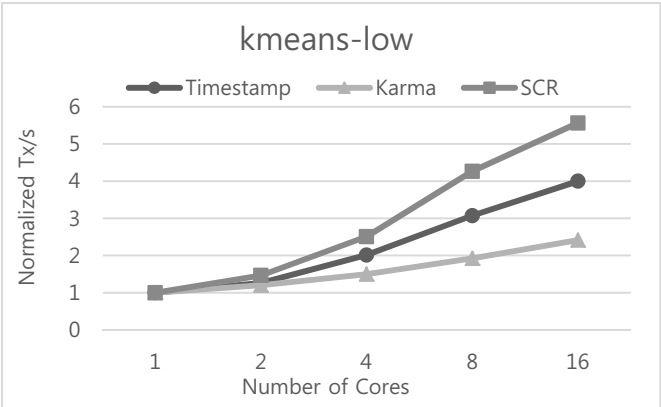
SCR 은 소프트웨어 트랜잭셔널 메모리 (STM) 인 TinySTM 위에 구현되었으며, 실험은 트랜잭셔널 메모리를 위한 시뮬레이션 세트인 STAMP 를 통해 진행되었다 [10][11][12]. 본 실험에서 SCR 의 임계 값은 10 으로 고정되어 진행되었다. 실험을 수행한 컴퓨터의 환경은 표 1 과 같다.

<표 1> 실험 환경

항목	설명
CPU	AMD Opteron Processor 6176 x 2 (2.30GHz x 12 x 2)
L1 Cache	64KB x 12 I-Cache, 64KB x 12 D-Cache
L2 Cache	512KB x 12
L2 Cache	12MB
Memory	128GB
OS	Red Hat Enterprise Linux Server release 6.0

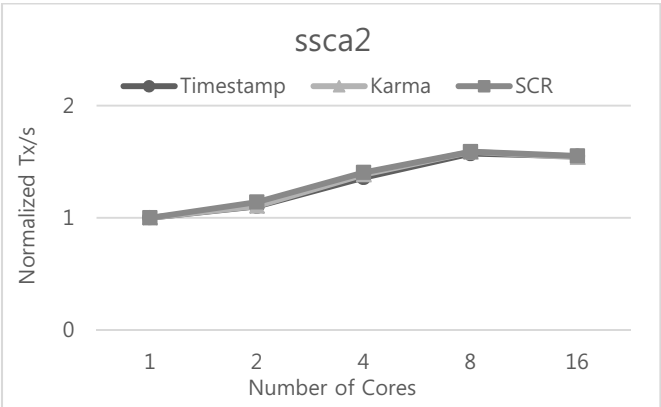
4.2. 실행 속도 비교

실험은 STAMP 어플리케이션을 각각 1, 2, 4, 8, 16 개의 코어를 사용하여, Timestamp, Karma, SCR 을 사용한 트랜잭셔널 메모리에서 각각 시뮬레이션을 진행하여 성능을 측정하였다.



(그림 3) kmeans-low contention 성능 결과

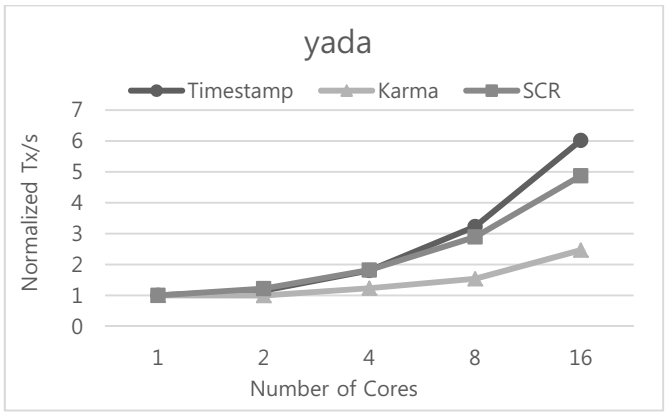
그림 3 은 SCR 이 기존의 충돌해결정책에 비해 성능이 가장 좋게 나온 kmeans-low contention 어플리케이션의 결과이다. 코어의 수가 적을 때는 어보트의 수가 적기 때문에 충돌해결정책이 적용되는 부분이 적어 큰 성능 향상을 보이지 못한다. 하지만 코어의 수가 증가함에 따라 공유자원의 충돌이 더욱 빈번하게 일어나고, 충돌해결정책의 역할이 중요해지면서 SCR 이 기존의 충돌해결정책보다 좋은 성능을 보임을 알 수 있다. SCR 은 Timestamp 에 비해 최대 39.3%, Karma 에 비해 최대 129.6%의 성능 향상을 보이며, SCR 이 기존의 두 충돌해결정책에 비해 성능 저하가 일어난 부분은 존재하지 않는다.



(그림 4) ssca2 성능 결과

그림 4 는 ssca2 어플리케이션의 결과이다. 이 어플리케이션에서는 기존의 충돌해결정책과 SCR 모두 오차 5%내외의 비슷한 성능을 보였다. ssca2 어플리케이션은 contention 수준이 매우 낮다. 그림 3 의 kmeans

역시 낮은 contention 수준을 갖지만, kmeans 는 100,000 ~ 1,000,000 의 충돌 횟수를 갖는 반면 ssca2 는 10 ~ 1,000 회의 매우 적은 충돌을 일으킨다. 즉, 공유 변수 충돌이 거의 일어나지 않았기 때문에 충돌해결 정책 자체가 큰 역할을 하지 못하였고, 결과적으로 어떤 정책을 사용하더라도 성능에 큰 영향을 미치지 못한 것이다.



(그림 5) yada 성능 결과

그림 5 는 SCR 이 가장 낮은 성능을 보인 yada 어플리케이션의 결과를 나타낸다. 이 어플리케이션에서는 SCR 이 Timestamp 에 전반적으로 성능이 낮으며, 최대 20%의 성능 저하를 보였다. SCR 이 Timestamp 에 비해 성능이 낮은 상황은 다음과 같다. 두 트랜잭션의 작업량 차이가 임계 값보다 커서 Karma 를 사용하기로 판단을 했지만, 두 트랜잭션의 시간차이가 작업량의 차이보다 크게 나는 상황이다. 위와 같은 상황에서는 작업량의 차이가 임계 값보다 커도 Timestamp 가 Karma 보다 유리하다.

이 외에도 세 충돌해결정책 중 SCR 이 두 번째로 좋은 성능을 보이는 어플리케이션은 존재하였지만, 셋 중에 가장 낮은 성능을 보인 경우는 존재하지 않았다. 반면 SCR 이 가장 좋은 성능을 보인 경우는 여러 어플리케이션에서 보였다. 즉, SCR 이 소수의 특정 상황에서는 Timestamp 등의 기존 충돌해결정책보다 성능이 하락될 수 있으나 가장 낮은 성능은 보이지 않으며, 많은 경우에서 가장 좋은 성능을 보인다고 할 수 있다. 결과적으로 전체 어플리케이션에서 평균 22%의 성능 향상을 보였다.

5. 결론

본 논문은 다중 코어 시스템에서 lock 을 대체할 동기화 기법으로 유망한 트랜잭셔널 메모리의 새로운 충돌해결정책을 제안하였다. 트랜잭셔널 메모리에서 충돌이 발생할 시, 어떤 스레드의 연산을 유효화 해줄 것인지에 대한 정책이 필요하다. 기존에는 먼저 생성된 트랜잭션 또는 작업량이 많은 트랜잭션을 선택해주는 방식 중 한 가지로 고정된 정책을 사용하였으나, 트랜잭션 단위의 동작에서 한 가지의 방식으로는 성능 저하를 일으킬 상황이 존재한다. 본 연구에서 제안하는 방식은 주어진 상황에 최적화된 충돌해결정책을 선택해주는 것이다. 판단 기준으로는 충돌한 두 트랜잭션의 작업량 차이를 사용하였다. 그 차이가 클 땐 작업량을 비교하여 결정하고, 차이가 작을 땐 트랜잭션이 생성된 시간을 비교하여 결정하는 방식이다. STAMP 실험 결과, SCR 은 기존의 충돌해결정책 대비 평균 22%의 성능 향상이 있었다.

본 연구 결과를 바탕으로, 향후에는 판단 기준이 되는 임계 값을 어플리케이션을 실행하는 중에 조정하여 더욱 유리한 판단을 할 수 있는 방법 등에 대한 연구를 진행할 계획이다.

참고문헌

[1] Maurice Herlihy, and J. Eliot B. Moss., “Transactional memory: Architectural support for lock-free data structures.” in *Proceeding of the 20th Annual International Symposium on Computer Architecture.* ACM, 1993.

[2] David Kanter, “Analysis of Haswell’s transactional memory.” *Real World Technologies* February 15, 2012.

[3] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. “Is transactional programming actually easier?” *ACM Sigplan Notices*. Vol. 45. No. 5. ACM, 2010.

[4] Richard M. Yoo, and Hsien-Hsin S. Lee. “Adaptive transaction scheduling for transactional memory systems.” *in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008.

[5] Geoffrey, Blake, Ronald G. Dreslinski, and Trevor Mudge. “Proactive transaction scheduling for contention management.” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009.

[6] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. scott. “A comprehensive strategy for contention management in software transactional memory.” in *Proceeding of the 14th ACM Sigplan symposium on Principles and practice of parallel programming.* ACM, 2009.

[7] William N. Scherer III, and Michael L. Scott. “Advanced contention management for dynamic software transactional memory.” in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005.

[8] Zhengyu He, Xiao Yu, and Bo Hong. “Profiling-based adaptive contention management for software transactional memory.” in *Proceeding of the Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012.

[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. “Dynamic performance tuning of word-based software transactional memory.” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.

[10] Chi Cao Minh, Jaewoong Chung, Kozyrakis C, Olukotun K., “STAMP: Stanford transactional applications for multi-processing.” *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008.

[11] Nir Shavit and Dan Touitou. "Software transactional memory." *Distributed Computing* 10.2 (1997): 99-116.