DOI: 10.3745/KIPSTA.2011.18A.1.019

다중 시그니처 비교를 통한 트랜잭셔널 메모리의 충돌해소 정책의 성능향상

김 덕 호[†] · 오 두 환[††] · 노 원 우[†††]

요 약

다중 코어 프로세서가 널리 보급되면서 멀티 쓰레디드 프로그램 상의 동기화를 용이하게 구현할 수 있는 해결 방안으로 트랜잭셔널 메모리가 각광을 받고 있다. 이를 위해 고성능의 하드웨어 트랜잭셔널 메모리에 관한 연구가 활발히 진행되고 있으며, 대표적인 연구결과로 UTM, VTM, FastTM, LogTM, LogTM-SE 등이 소개되었다. 특히, 충돌 감지 정책으로 시그니처를 사용한 LogTM-SE는 효율적인 메모리 관리와 쓰레드 스케쥴링을 통해 고성능의 트랜잭셔널 메모리를 구현하였다. 하지만, 이 방식은 프로세서 내부의 코어 수가 증가하는 것에 비례하여 한 코어가 비교해야 하는 시그니처의 수가 증가하는 문제점을 갖고 있다. 이는 시그니처 처리 과정에서 병목현상을 야기하여 전체 성능을 저해하는 요인이 될 수 있다. 본 논문에서는 시그니처 비교 과정에서 나타날 수 있는 이러한 병목 현상을 개선하여 전체 트랜잭셔널 메모리의 성능 향상을 이루고자 다중 시그니처 비교 방식의 새로운 구조를 제안한다.

키워드 : 트랜잭셔널 메모리, 충돌 감지 정책, 병렬 프로그래밍, 멀티 쓰레딩, 다중 코어 프로세서

Multiple Signature Comparison of LogTM-SE for Fast Conflict Detection

Deokho Kim[†] · Doohwan Oh[††] · Won W. Ro[†††]

ABSTRACT

As era of multi-core processors has arrived, transactional memory has been considered as an effective method to achieve easy and fast multi-threaded programming. Various hardware transactional memory systems such as UTM, VTM, FastTM, LogTM, and LogTM-SE, have been introduced in order to implement high-performance multi-core processors. Especially, LogTM-SE has provided study performance with an efficient memory management policy and a practical thread scheduling method through conflict detection based on signatures. However, increasing number of cores on a processor imposes the hardware complexity for signature processing. This causes overall performance degradation due to the heavy workload on signature comparison. In this paper, we propose a new architecture of multiple signature comparison to improve conflict detection of signature based transactional memory systems.

Keywords : Transactional Memory, Conflict Detection, Parallel Programming, Multi-Threading, Multi-Core Processor

1. 서 론

컴퓨터 시스템은 트랜지스터 집적도의 증가에 비례하여 그 성능이 비약적으로 향상되어 왔다. 하지만, 최근 들어 집적도의 증가에도 불구하고 프로세서의 동작 속도가 향상되지 못하는 한계에 부딪쳤다. 즉, 더 이상의 빠른 동작 주파수는 오히려 과도한 전력 소모 및 발열 등의 물리적 한계를 야기하였다. 이에 따라, 단일 코어의 동작 속도 향상이 한계에 부딪쳤고, 최근의 CPU 제조업체들은 단일 코어의 성능 향상에 대한 연구에서 방향을 전환하여 여러 개의 코어를 한 CPU안에 집적하는 멀티코어 시스템의 개발에 집중하게 되었다[1].

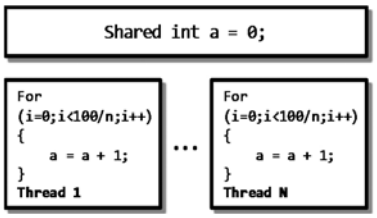
한편, 응용 프로그램의 경우, 과거에는 동작 주파수의 발전에 따라 손쉽게 그 성능 향상을 이룰 수 있었다. 즉, 소프트웨어는 알고리즘 수정이나 프로그램의 재개발과 같은 노력 없이도 하드웨어 발전의 도움으로 성능 향상을 이룰 수 있었으며, 이는 또한 점차 복잡한 소프트웨어들이 등장하게 되는 발판을 마련해 주었다.

하지만, 더 이상의 동작 주파수 증가가 어려운 지금의 상황을 고려해 볼 때, 기존의 소프트웨어들이 그 성능 향상을 이루기 위하여서는 다중 코어의 활용이 매우 중요하다. 즉, 효율적

※ 이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2010-0013202).
† 준 회 원 : 연세대학교 전기전자공학과 석사과정
†† 준 회 원 : 연세대학교 전기전자공학과 박사과정
††† 종신회원 : 연세대학교 전기전자공학과 조교수
논문접수 : 2010년 9월 13일
수 정 일 : 1차 2010년 11월 2일, 2차 2010년 11월 24일
심사완료 : 2010년 11월 25일

인 병렬화를 활용한 프로그램을 이용하여, 최근의 다중 코어 프로세서의 컴퓨팅 파워를 온전히 활용하는 것이 매우 중요하다. 하지만, 이러한 병렬 프로그램은, 순차적인 프로그램에 비해 설계가 복잡하고, 디버깅 과정이 어렵다는 단점이 있다.

기존의 프로그램과는 다르게 다중 코어 프로세서에서 동작하는 병렬 프로그래밍은 데이터의 동기화(Synchronization)에 대한 고려가 반드시 이루어져야 한다. 동기화는 병렬화된 프로그래밍에서 가장 중요하게 고려해야 할 사항이며, 병렬 프로그래밍을 어렵게 하는 요소 중 하나이다.



(그림 1) 동기화 오류의 예

(그림 1)은 일반적으로 다중 코어 프로세서에서 일어날 수 있는 병렬 프로그램 (또는 멀티 쓰레디드 프로그램) 의 오류를 보여주는 예시이다. 어떤 변수 a에 1을 백 번 더하는 프로그램을 멀티 쓰레드로 만들게 되면 (그림 1)과 같은 구조를 가지게 된다. n개의 thread가 존재할 경우 $100/n$개 만큼의 덧셈을 각 thread에 분배해서 수행할 수 있게 된다. 이 때, 여러 개의 쓰레드가 동시에 동작하는 것이 가능하여, 한 개의 쓰레드가 a를 읽는 사이에 다른 쓰레드가 a를 쓰게 될 경우가 발생할 수 있다. 즉, a를 읽었던 쓰레드는 변화된 값을 인지하지 못하고 잘못된 데이터 값을 덮어쓰게 된다. 그 결과 최종적으로 a 값은 우리가 의도했던 100보다 작은 값을 갖게 된다.

현재는 위와 같은 오류를 막기 위해 전통적인 데이터 동기화 방식인 lock을 병렬 프로그램에 적용하고 있다. 즉, 지정된 구간에 lock을 적용하여 이 구간에 포함된 코드들이 배타적으로 수행되도록 제한을 두어 데이터의 일관성을 보장하는 방식이다. 하지만 lock방식은 오직 하나의 쓰레드만 지정된 구간 안에 진입하고 나머지 쓰레드는 진입한 쓰레드가 해당 구간의 명령어를 모두 수행할 동안 대기해야 한다는 제한이 있다. 따라서, 다중 코어에서 lock방식의 병렬 프로그램이 수행될 경우 프로그래머가 지정한 배타적 구간에서는 오직 하나의 코어만 운영되는 성능상의 치명적 단점을 가지게 된다.

트랜잭셔널 메모리는 위와 같은 lock의 성능 저하 문제를 개선하기 위해 고안되었다. 트랜잭셔널 메모리는 배타적 구간에서도 다중(n개)의 쓰레드가 수행될 수 있도록 구현함으로써 lock에 비해 최대 n배에서 최소 1배 이상의 성능을 기대할 수 있게 된다. 또한, lock이 없이 프로그래밍이 가능하므로 lock을 통해 나타날 수 있는 livelock이나 deadlock과 같은 오작동에 대한 위험을 피할 수 있으며 쓰레드의 동기화 문제를 트랜잭셔널 메모리가 관리하므로 프로그래머는 좀 더 수월하게 병렬 프로그래밍을 구현할 수 있는 장점을 가진다.

최초의 트랜잭셔널 메모리는 Herlihy 등에 의해 데이터베이스의 관리 시스템에 기초한 하드웨어와 소프트웨어의 조합으로 만들어져 발표됐다. 현재는 고성능을 목표로 하는 하드웨어 트랜잭셔널 메모리 위주로 트랜잭셔널 메모리 연구가 진행되고 있다[2]. 하드웨어 트랜잭셔널 메모리는 데이터 관리 정책과 충돌 감지 정책을 개선한 LogTM[3]과 LogTM-SE[4]가 2006년과 2007년에 각각 발표되면서 더욱 큰 주목을 받게 되었다. 특히 시그니처를 사용하여 다중 쓰레드간 공유 데이터의 충돌을 감지하는 시스템을 새롭게 구현한 LogTM-SE의 효용성이 큰 주목을 받고 있다. 하지만 이 방식은 코어의 수가 증가할수록 시그니처 비교에 대한 작업량이 증대되어 성능 저하를 야기할 수 있는 문제점을 내포하고 있다. 따라서, 본 논문에서는 LogTM-SE에 제시된 시그니처 비교기를 개선하여 다중 코어 프로세서에서 나타날 수 있는 성능 저하를 개선하였다.

본 논문의 구성은 다음과 같다. 2장에서는 트랜잭셔널 메모리의 개념과 시그니처에 대하여 설명한다. 3장에서는 새롭게 개선된 다중 시그니처 비교 방안에 대해 설명한다. 4장에서는 개선된 다중 시그니처 비교를 통한 시뮬레이션 결과 및 분석에 대해 서술한다. 마지막으로 5장에서는 결론에 대해 기술한다.

2. 트랜잭셔널 메모리

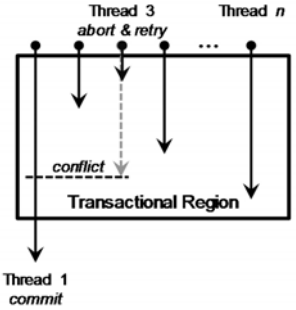
2.1 트랜잭셔널 메모리와 두 가지 정책

트랜잭셔널 메모리는 기존의 lock이 배타적으로 코드를 수행하는 것과는 달리 인스트럭션 단위의 atomic한 수행을 보장함으로써 배타적인 구간 내에서도 동시에 여러 쓰레드가 실행될 수 있는 구조를 가진다. 이를 통해 lock 보다 우수한 병렬 프로그래밍 성능을 발휘할 수 있게 된다. 이 때 발생할 수 있는 문제점들은 데이터 관리 정책과 충돌 감지 정책을 통하여 해결할 수 있다.

트랜잭셔널 메모리는 기존의 배타적 구간을 트랜잭션 구간으로 지정하여 최대한 모든 쓰레드가 구간 내에서 막힘없이 진행되도록 한다. 이 때 충돌 감지 정책은 데이터의 일관성을 위배하는 경우가 발생하는지를 감시하게 된다. 만약 데이터의 일관성에 위배되는 동작이 감지되면 충돌(conflict) 의 가능성을 해당 쓰레드에 알려 구간 내에서 수행했던 모든 명령어들을 취소하고 구간의 시작 지점으로 되돌아가게 한다. 이 때 다시 돌아가는 것을 *abort*라고 하며 *abort*가 된 쓰레드가 시작점에서 문제없이 재실행 될 수 있도록 데이터를 관리하는 것이 데이터 관리 정책이다.

(그림 2)는 트랜잭션 영역에서 여러 쓰레드가 동작하는 구조를 나타낸 것이다. n개의 쓰레드가 존재할 때, 쓰레드 1이 충돌 없이 수행을 끝마칠 경우 *commit*이 되며 트랜잭션 영역을 무사히 벗어나게 된다. 구간 안에 존재하는 각 쓰레드는 매 load와 store마다 트랜잭션이 충돌하는지 검사를 하게 되고 충돌이 없을 경우 계속 진행하여 *commit*할 수 있게 된다. 반면에 쓰레드 3은 어느 정도 진행한 뒤 쓰레드 1과 충돌을 일으키게 된다. 이 때, 쓰레드 3은 *abort*되어 트

랜잭션 구간에서 사용한 데이터를 모두 복구한 후 구간의 시작점으로 돌아가 명령어들을 재실행하게 된다. 쓰레드 3을 제외한 모든 (n-1)쓰레드에서 더 이상 충돌이 일어나지 않을 경우 나머지 쓰레드는 *abort*없이 모두 *commit*된다. *Abort*가 되어 다시 수행하는 쓰레드가 적으면 적을수록 반복되어 수행되는 명령어가 줄어들게 되므로 트랜잭셔널 메모리의 성능은 증가하게 된다. 어떤 쓰레드가 *abort*될 지를 결정하고 수행하는 것은 충돌 감지 정책과 데이터 관리 정책에 위해서 정의된다. 따라서 두 정책은 트랜잭셔널 메모리의 성능을 결정하는 핵심 기술이라 볼 수 있다.



(그림 2) 트랜잭셔널 메모리의 기본 동작

데이터 관리 정책은 *abort*가 된 경우 트랜잭션 영역을 복구하기 위해 데이터를 보관하는 것에 대한 정책이며, 충돌 감지 정책은 사용하는 데이터가 다른 쓰레드에 의해 이미 사용되었는지를 감지하여 데이터의 동기화 이상 유무를 판단하여 쓰레드간 공유 데이터의 충돌을 감지하는 정책이다. 데이터 관리 정책과 충돌 감지 정책은 각기 eager와 lazy 두 가지 특성으로 구현될 수 있다. 따라서 기본적으로 총 4개의 조합을 가지는 트랜잭셔널 메모리를 구성할 수 있다[3].

데이터 관리 정책은 트랜잭션 영역에 들어갈 때에 존재하던 초기 데이터를 어떤 방식으로 보관하느냐에 따라서 eager version management와 lazy version management로 나뉘게 된다. Eager version management는 in-place에 바뀐 데이터를 직접 쓰고, 초기 데이터를 버퍼에 저장하여 *abort*가 일어나지 않을 경우 빠르게 *commit*할 수 있는 환경을 제공한다. 이 때, in-place는 실제 데이터가 사용되고 보관되는 곳을 나타내며 다중 코어 프로세서에서는 각 프로세서에 속한 L1 cache를 지칭한다. Eager version management와는 반대로 lazy version management는 바뀐 데이터를 버퍼에 저장하고 초기 데이터를 in-place에 저장하여 *abort*시에 데이터를 복구하는 과정 없이 빠르게 retry 할 수 있도록 한다. 그러나 *commit*하게 될 경우, 버퍼에서 데이터를 in-place로 복사하여야 하므로 빠른 *commit*을 할 수 없는 단점이 있다.

충돌 감지 정책은 각 프로세서가 읽거나 쓰는 데이터가 다른 쓰레드에 의하여 사용되었는지를 분석하여 *abort*의 여부를 결정하는 역할을 한다. Eager conflict detection은 매 load나 store가 나타날 때 마다 다른 코어에서 read나 write을 하였는지 측정하여 충돌을 감지하고 *abort*를 결정하게 된다. 트랜잭션 구간에서 매 load나 store마다 충돌을 확인하는 것과 달리 lazy conflict detection은 각 쓰레드가 트랜잭션 영역을 다 수행한 뒤, 마지막 *commit* 되기 전에 구간 내에서 다른 쓰레드와 충돌이 없었는지를 확인하여 *abort*할지를 판단한다. 따라서, 빠른 *commit*이 가능하지만 large 컨텐션을 가진 프로그램의 경우에는 빠른 *abort* 가 필요하므로 성능이 저하될 수 있다.

2.2 시그니처를 사용한 충돌 감지 정책

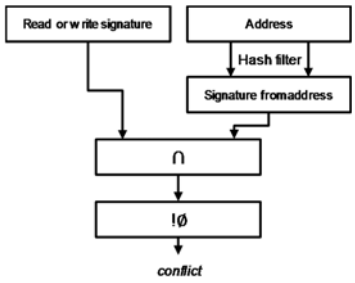
L. Yen 등이 발표한 LogTM-SE(LogTM-Signature Edition)[4] 에서는 충돌 감지를 위해 기존의 캐시 구조를 바꾸던 것에서 탈피하여 독립적인 하드웨어를 이용하여 충돌을 감지할 수 있는 시스템을 구현하였다. LogTM-SE는 시그니처를 도입하여 캐시 구조의 변형 없이, 충돌감지를 구현하였고, 써머리 시그니처를 사용하여 컨텍스트 스위칭이나 페이징에대한 문제를 해결하여 보다 효율적으로 동작할 수 있는 트랜잭셔널 메모리를 구현하였다[4].

LogTM-SE의 시그니처는 L. Ceze 등이 발표한 Bulk Disambiguation of Speculative Threads in Multiprocessors[5] 의 시그니처 개념에서 도입되었다. 시그니처는 load나 store가 참조하는 address를 블룸 필터를 통과시켜 나온 해시 값의 집합으로 이루어 진다. 이 때 생성되는 해시 값은 트랜잭션이 끝나거나 *abort*될 때까지 지속적으로 시그니처에 저장되며 서로 다른 쓰레드의 시그니처와 비교를 통해 쓰레드간 충돌을 감지한다.

기본적으로 프로세서내의 한 코어에서 load나 store를 수행하게 되는 경우 그 주소를 캐시 프로토콜을 통하여 다른 코어에 전달하게 된다. 이때, 주소를 전달받은 코어는 블룸 필터를 통과하여 얻어진 해시 값을 각 코어에 저장된 read 시그니처 및 write 시그니처와 비교하여 일치하는 주소가 있는지를 검사한다. 만약, 이 검사에서 일치하는 주소가 존재하면 해당 쓰레드는 충돌임을 감지하게 된다.

충돌 감지 정책에서 load에 사용되는 주소가 전송되어 다른 코어에 도착할 경우 오직 그 코어의 write 시그니처와 비교하여 충돌 여부를 판단하면 된다. 반면에 store를 통해 전달 받은 주소는 read 시그니처와 write 시그니처 모두와 비교하여 둘 중 하나의 시그니처에서 해당하는 주소가 존재하면 충돌로 판단하게 된다.

(그림 3)은 코어 내부에서 시그니처 비교를 통해 충돌을 감지하는 시스템을 도식화한 것이다. 다른 코어에서 전송받은 주소 값을 블룸 필터를 통과시켜 시그니처 형태로 출력한 후 코어 내부에 저장되어있던 시그니처와 교집합 연산으로 비교를 하게 된다. 이 때, 얻어진 결과값의 공집합여부를 검사하여 충돌 여부를 판별하게 된다. 만약 결과값이 공집합일 경우 해당 주소에 대한 충돌은 발생하지 않는 것이 된다. LogTM-SE에서는 eager version management 와 eager conflict detection을 사용하므로 모든 코어의 load 및 store명령에서 위와 같은 시그니처 비교 작업이 이루어 질



(그림 3) 시그니처를 통한 충돌 감지

수 있으며, 충돌이 발생하면 즉시 해당 쓰레드를 *abort*하여 충돌의 위험을 제거하게 된다.

3 장에서는 시그니처 비교기 구현의 자세한 사항과, 본 논문에서 제시하는 새로운 구조에 대해 서술한다.

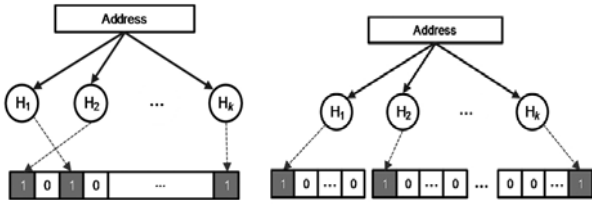
3. 병렬 해시 필터와 다중 시그니처 비교

본 장에서는 시그니처 비교기 구현의 자세한 사항과 본 논문에서 제시하는 다중 시그니처 비교기의 구조에 대하여 설명한다.

3.1 병렬 해시 필터

D. Sanchez 등은 [6]에서 k개의 해시 함수를 통한 시그니처 비교기의 구현에 대하여 연구하였다. 이 논문에서 제시한 시그니처 비교기는 다중의 해시 함수를 통해 구현되었으며 그 종류는 True Bloom Signatures와 Parallel Bloom Signatures가 존재한다.

True Bloom Signatures는 가장 기본적인 해시 함수를 사용하였으며, 이들 함수를 통해 얻어진 결과값을 바탕으로 시그니처를 구현하여 비교를 하는 방식이다. 기존의 시그니처는 n개의 해시 함수를 통해 나온 n개의 결과를 통해 시그니처의 특정한 n개의 위치를 1로 기록함으로써 해당 주소를 시그니처에 기록하게 된다. 따라서 시그니처의 비교는 n개의 비트가 모두 일치하는지를 검사하여 해당하는 주소가 시그니처에 포함되는지를 판별하게 된다.



(a) True Bloom Signatures (b) Parallel Bloom Signatures

(그림 4) Various Bloom signatures

(그림 4(a))는 특정 주소 값이 k개의 해시 함수를 통과한 후 어떻게 시그니처에 기록되는지를 보이고 있다. 이렇게 생성된 시그니처는 다른 시그니처와 선택된 비트만을 비교하여 충돌을 감지하게 된다. k개의 비트가 일치하는 경우에만 주소 값이 시그니처 안에 포함되어 있다고 판단할 수 있다. [6]에서 D. Sanchez 등은 이러한 구조를 발전시키고자 각 해시 함수에 연결된 시그니처 비트의 수를 줄여 적은 포트 만으로 구현이 가능한 Parallel Bloom Signatures를 소개하였다. (그림 4(b))에서처럼 각 해시 함수가 저장할 수 있는 비트 필드가 특정 구간으로 한정되는 구조를 사용하여 더욱 간단해진 시그니처 생성 구조를 구현하였다. 따라서, 한 시그니처 안에 나누어진 각 비트 필드는 서로 독립적으로 관리되며, 각 구간은 하나의 주소에 대하여 오직 하나의 비트만을 1로 기록하게 된다.

3.2 시그니처 비교의 문제점 및 다중 시그니처의 구조

본 절에서는 기존의 시그니처 비교가 발생시킬 수 있는 문제점과 다중 시그니처의 구조에 대해 설명한다.

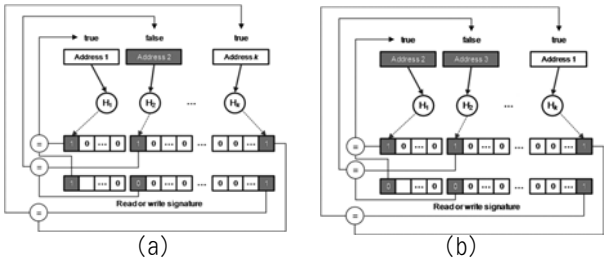
트랜잭셔널 메모리는 다중 멀티코어 프로세서에서 병렬화된 프로그램의 성능향상을 위해 제안되었다. 어떤 코어에서 load나 store를 수행할 경우 캐시 프로토콜을 통하여 해당 주소 값을 다른 코어들에게 전송하게 된다. 주소 값을 받은 코어들은 자신의 시그니처와 이 주소 값을 비교하여 충돌 여부를 판별하고 얻어진 결과를 응답으로 보내게 된다. 이 충돌 감지 정책은 매 load나 store를 수행할 때마다 캐시 프로토콜을 통하여 데이터를 지속적으로 주고 받게 되므로 데이터의 신뢰성을 보장받는 것은 매우 복잡한 작업이다. 특히, 네트워크로 구성된 다중 코어 프로세서의 경우 코어 간 데이터 전송에 대한 지연시간이 추가적으로 소요되며 코어의 응답시간을 기다릴 경우 수 사이클이 더 소요될 수 있다. 기존의 논문에서는 하나의 시그니처를 복사하고 비교하는 것은 큰 부하가 걸리지 않는 것으로 가정해왔다[4].

한 개의 코어를 기준으로 하였을 때, 시그니처의 비교와 그 결과를 재전송 받는데 소요되는 시간이 큰 부하를 야기하지는 않지만, 다중 프로세서에서 코어의 개수가 늘어남에 따라 한 코어가 비교해야 하는 주소 값들의 개수 역시 늘어나게 되므로 큰 부하를 야기할 수 있다. 따라서, 코어 수에 비례하여 시그니처의 비교에 따라 발생하는 지연시간은 점차로 증가하게 된다. 게다가 트랜잭션 구간을 거의 다 수행한 쓰레드의 경우 *commit*이 가까워질수록 많은 데이터의 주소 값을 갖게 된다. 그 결과 많은 코어로부터 충돌 감지에 대한 요청이 들어오게 되고, 한 코어에서 한번에 비교해야 하는 시그니처의 개수가 점차로 늘어나게 된다. 따라서 본 논문에서는 시그니처 비교에서 오는 성능 저하를 해결하고자 적은 수의 시그니처와 많은 수의 시그니처를 효율적으로 비교할 수 있는 다중 시그니처 비교의 구조를 제안한다.

Parallel Bloom Signatures를 사용할 경우 시그니처의 일정 부분인 각 비트 필드는 하나의 해시 함수에 종속되어 하나의 주소 값에 대해 오직 한 특정 비트만을 설정하게 된다. 이러한 구조에서 각 해시 함수에 여러 개의 주소 값을 넣었을 경우 각 비트 필드들은 서로 다른 주소 값의 해시 결과를 갖게 된다. 따라서, 하나의 시그니처 안에 존재하는 여러

비트 필드 중에서 단 한 개의 비트 필드라도 시그니처 비교를 통과하지 못하는 경우 그 주소 값은 conflict가 아니므로 더 이상 비교할 필요가 없게 된다. 이는 여러 개의 주소를 한 번에 비교할 수 있는 구조로 발전될 수 있다.

(그림 5)는 다중 시그니처 비교의 동작 구조를 나타낸 그림이다. 이 구조는 각 해시 함수에 서로 다른 주소 값을 넣어 비트 필드를 개별적으로 비교할 수 있도록 설계된 방식이다. 이 때 모든 비트 필드의 결과값이 참이 될 경우만 충돌로 판별되므로 (그림 5(a))에서처럼 address2가 단 한번 *false*를 얻었을 경우라도 address2는 conflict가 아님을 확인할 수 있다. (그림 5(b))는 현재 결과가 *true*일 경우 주소 값을 왼쪽으로 이동하면서 해시 함수를 달리해 비교를 계속 수행하는 것을 나타낸다. 모든 address에 대해 *false*가 발생하거나 *k*번의 비교가 이루어진 뒤에는 주소 값의 비교가 완료 된다.



(그림 5) 다중 시그니처 비교

한 번 시그니처를 비교하는데 한 사이클이 걸린다고 할 때, *k*개의 주소 값이 들어왔을 경우 *k*번째의 주소 값은 *k-1* 사이클을 대기한 후에 비교할 수 있지만 다중 시그니처 비교를 이용할 경우 최소 한 사이클 만에 *k*개의 주소에 대한 *conflict* 여부를 파악할 수 있다. 또한, 최대 지연시간은 *k* 사이클 이므로 원래의 지연 시간과 같아지게 된다. 부가적으로 간단한 계산을 통하여 전달받은 주소 값이 *k*개보다 작은 경우 기존의 단일 시그니처 비교 방식을 사용하여 *k*보다 적은 개수의 주소를 *k* 사이클에 나누어 비교하는 것을 방지하였다.

4 장에서는 실험 결과 및 분석을 통하여 본 논문에서 제시한 다중 시그니처 비교가 트랜잭셔널 메모리의 성능에 미치는 영향을 분석하고 코어의 개수에 따라 얼마나 큰 성능 향상을 보이는지를 서술하였다.

4. 실험 결과 및 분석

본 장에서는 다중 시그니처 비교의 성능을 검증하고자 벤치마크를 통해 기존의 방식과 새롭게 제안한 방식의 시그니처 비교 사이클 수를 측정하여 그 성능을 분석하였다. 또한, 매 사이클에 들어오는 주소 값의 수를 측정하여 다중 시그니처 비교기가 사용되는 빈도를 함께 보였다.

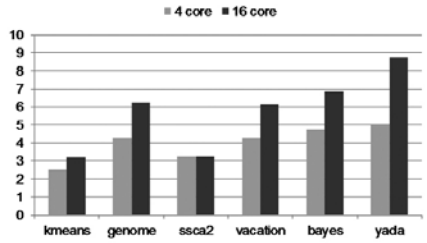
트랜잭셔널 메모리의 시뮬레이션은 GEMS[7] 와 Simics [8]를 통해 구현되었으며, 사용한 프로세서의 코어 수는 4개와 16개로 설정하여 검증을 진행하였다. 구현된 다중 시그니처 비교기의 성능을 평가하기 위해서 트랜잭션의 성능평가에 주로 사용되는 STAMP[9] 벤치마크를 사용하였다.

본 논문에서는 4개의 해시 함수를 사용하여 들어온 주소 값이 4개를 넘는 경우에만 다중 시그니처 비교를 하도록 설계하여 성능향상을 이끌었으며, 그 외에는 기존의 방식을 그대로 사용하여 다중 시그니처 비교가 야기할 수 있는 성능 하락을 방지하였다.

〈표 1〉 STAMP 벤치마크

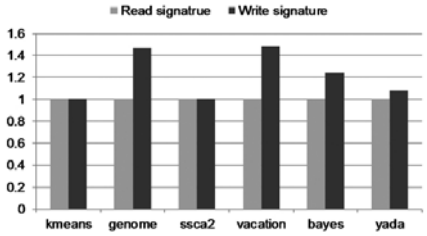
Application	Tx Length	R/W set	Tx Time	Contention
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
kmeans	Short	Small	Low	Low
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Low/Medium
yada	Long	Large	High	Medium

<표 1>은 C. Minh 등이 저술한 [9]를 참조하여 STAMP 벤치마크 어플리케이션들의 성격에 대해 정리한 것이다. Tx는 트랜잭션을 의미하며 각 어플리케이션은 트랜잭션의 길이와, Read와 Write set의 수, 트랜잭션 수행시간, 그리고 컨텐션 등으로 나타내어진다. 이에 따라 한 주소 값에 대해 얼마나 많은 쓰레드가 접근을 하게 되는지에 대한 정도를 분석할 수 있게 된다. 즉 컨텐션이 높고, 트랜잭션이 크고 트랜잭션 수행시간이 길수록 시그니처 비교 회수가 증가하게 된다.



(그림 6) 코어 수에 따른 최대 시그니처의 개수

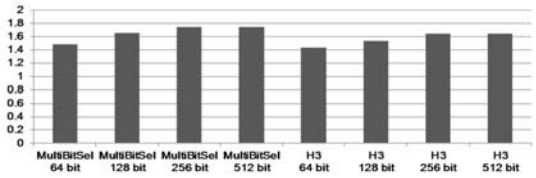
(그림 6)은 코어 수가 변할 때 한 사이클에 전달되는 최대 시그니처의 개수들의 평균이다. 컨텐션이 많은 어플리케이션일수록 동시에 비교해야 하는 최대 시그니처의 개수가 증가하며, 코어 수가 많을수록 더욱 큰 값을 갖게 된다.



(그림 7) 시그니처 비교 스피드 업

(그림 7)은 MultiBitSel Parallel 해시 함수를 사용했을 경우 4 코어 프로세서에서의 64비트의 read시그니처와 write 시그니처 비교기의 성능이다. 기존의 비교기가 한 시그니처를 비교하는 시간을 1이라고 놓았을 때, 본 논문에서 제시

하는 방식을 통해 줄어든 비교시간을 스피드 업으로 나타낸 그래프이다. 시그니처가 동시에 들어온다 하더라도 write 시그니처에 비해 read 시그니처의 성능향상이 거의 없는 것으로 나타난다. 특히, 트랜잭션이 크고 컨텐션이 많은 어플리케이션에 대해서 write 시그니처 비교 성능은 약 1.5배까지 증가하는 것을 볼 수 있다. 이는 load 나 store명령으로 발생된 주소 값이 항상 write 시그니처와 비교가 되는 것과는 달리 read 시그니처는 store명령에 의해 비교가 이루어질 때만 사용된다. 따라서 write 시그니처에 대한 성능 향상이 상대적으로 훨씬 크게 나타나게 된다.



(그림 8) 해시 함수와 시그니처에 대한 성능비교

(그림 8)은 vacation 어플리케이션을 사용하여 LogTM-SE에 사용하는 두 가지 해시 함수 MultiBitSel과 H3 함수의 parallel 버전에 대해서 시그니처의 크기를 변화하며 다중 시그니처 비교기의 성능을 관찰한 것이다. 이 실험 결과를 바탕으로 MultiBitSel 해시 함수가 256비트 이상의 시그니처를 사용할 경우 본 논문에서 제시하는 다중 시그니처 비교기가 가장 좋은 스피드 업을 보이는 것을 관찰할 수 있다.

5. 결 론

본 논문에서는 새로운 다중 시그니처 비교기를 제안하여 트랜잭셔널 메모리의 성능향상을 이루었다. 특히 코어의 개수가 늘어날수록 동시에 비교해야 하는 시그니처의 수가 증가하므로 시그니처 비교기의 성능이 전체 시스템에서 큰 비중을 차지하는 것을 볼 수 있었다. 다중 시그니처 비교기를 통해 컨텐션이 큰 어플리케이션에 대해서 비교기의 성능을 약 1.5배 이상 향상되었으며 다중 시그니처 비교기를 위한 최적의 해시 함수와 시그니처의 크기 또한 정의하였다.

참 고 문 헌

[1] D. Geer, “Chip makers turn to multicore processors,” Computer, vol. 38, no. 5, pp. 11-13, may 2005.
[2] M. 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.
[3] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill and D.A. Wood, "LogTM: log-based transactional memory," High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, pp. 254-265, 11-15 Feb. 2006.
[4] 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, pp. 261-272, Feb. 2007.
[5] L. Ceze, J. Tuck, C. Cascaval and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," Computer Architecture, 2006. ISCA '06. 33rd International Symposium on, pp.227-238, 2006.
[6] D. Sanchez, L. Yen, M.D. Hill and K. Sankaralingam, "Implementing Signatures for Transactional Memory," Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on, pp.123-133, 1-5 Dec. 2007.
[7] 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, pp. 92-99, Sept. 2005.
[8] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. IEEE Computer, vol. 35, no. 2, pp. 50-58, Feb. 2002.
[9] 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, pp. 35-46, 2008.

김 덕 호

e-mail : nautes87@yonsei.ac.kr
2010년 연세대학교 전기전자공학과(학사)
2010년 연세대학교 전기전자공학과 석사과정
관심분야 : 컴퓨터 시스템, 병렬 프로세싱, Cell 프로세서 등

오 두 환

e-mail : ohdooh@yonsei.ac.kr
2007년 경희대학교 전자공학과(학사)
2007년~2009년 한국전자부품연구원 연구원
2010년 연세대학교 전기전자공학과(석사)
2010년 연세대학교 전기전자공학과 박사과정
관심분야 : 컴퓨터 시스템, 트랜잭셔널 메모리 등

노 원 우

e-mail : wro@yonsei.ac.kr
1996년 연세대학교 전기공학과 졸업(학사)
1999년 University of Southern California(석사)
2004년 University of Southern California (공학박사)
2003년~2004년 Apple Computer Inc. 인턴 연구원
2004년~2007년 California State University 전기 및 컴퓨터 공학과 조교수
2006년~2007년 ARM Inc. 소프트웨어 엔지니어
2007년~현 재 연세대학교 전기전자공학과 조교수
관심분야 : 고성능 마이크로프로세서 디자인, 컴파일러 최적화, 임베디드 시스템 디자인 등