제33회 한국정보처리학회 춘계학술발표대회 논문집 제17권 제1호 (2010. 4)

동적 코드변환 기술을 이용한 소프트웨어 트랜잭션 메모리 기법 설계*

이동우, 김지홍, 엄영익
성균관대학교 정보통신공학부
e-mail:{cowsboys, jji-long ,yieom}@ece.skku.ac.kr

Design of Software Transactional Memory by Binary Translation

Dong-woo Lee, Jee Hong Kim, and Yong Ik Eom
School of Information and Communication Engineering, Sungkyunkwan Univ.

요 약

최근 프로세서가 코어 개수를 늘리는 구조로 발전함에 따라 병렬프로그래밍의 중요성이 더욱 강조되고 있다. 병렬프로그래밍에서 발생하는 공유자원에 대한 경쟁조건을 제어하기 위한 효율적인 방법으로 여러 가지 락-프리 동기화 기법이 제안되어 왔다. 그 중 소프트웨어 트랜잭션 메모리는 지금까지 하드웨어적인 방법과 소프트웨어적인 방법 등 여러 가지 방법으로 구현되었지만 여러 가지 하드웨어적인 제약과 기존의 소스코드를 수정해야 하는 문제점이 있다. 이러한 문제를 해결하기 위해 본 논문에서는 동적 코드 변환기술을 이용한 소프트웨어 트랜잭션 메모리 기법을 제안하고 기존 구현과 비교평가하였다.

1. 서론

최근 프로세서 개발 동향이 동작 속도(clock)를 높이는 데 한계를 느끼고 코어의 수를 늘리는 것이 중심이 되었다. 이전까지는 프로그램을 빠르게 실행하기 위해서는 새로운 프로세서를 사용하면 되었지만 이제는 그렇지 않다. 새로운 프로세서 환경에서 빠른 프로그램을 작성하기 위해서는 병렬프로그래밍을 해야 한다[4]. 하지만 병렬 프로그램은 동작 과정을 예측하기 힘들기 때문에 병렬 프로그램을 작성하는 것은 매우 어렵다[3][6].

병렬 프로그래밍에서 맞닥뜨리게 되는 가장 큰 문제는 공유 자원을 여러 쓰레드에서 동시에 접근하면서 생기는 경쟁 조건(race condition)이다. 이러한 경쟁 조건을 제어하기 위해 뮤텍스(MUTEX)나 세마포어(semaphore)와 같은 락-기반(lock-based)의 동기화 기법을 사용한다. 프로그램이 경쟁 조건을 배제하고 정확성을 확보하기 위해 사용하는 락 기반 동기화 기법은 성능과 확장성을 많은 부분 희생해야 한다. 또한 락을 이용한 동기화 기법은 데드-락(dead-lock), 우선순위역전(priority inversion)등의 문제점을 가지고 있기 때문에 사용에 있어서 많은 주의가 필요하다.

이러한 락 기반 동기화 기법의 단점을 해결하기 위해 제안된 방법이 락-프리(lock-free) 동기화 기법이다. 락-프리 동기화 기법들은 뮤텍스나 세마포어처럼 일반화 할 수 없지만 락을 기반으로 하는 동기화 기법에서 발생하는 문제점을 근본적으로 해결 할 수 있기 때문에 락-기반의 동기화 기법의 대안으로 주목 받고 있다. 소프트웨어 트랜잭션 메모리(software transactional memory, STM)[2]는 락-프리 동기화 기법의 대표적인 방법이다.

소프트웨어 트랜잭션 메모리를 구현하는 방법은 하드웨어적인 방법과 소프트웨어적인 방법 등 매우 다양하다. 하지만 기존 방법들은 여러 가지 하드웨어적인 제약과 기존의 함수를 각 구현에서 제공하는 API를 통해 수정해야 하는 문제점이 있었다. 이러한 문제를 해결하기 위해 본 논문에서는 유연하고 성능 좋은 소프트웨어 트랜잭션 메모리 구현을 위해 동적 코드변환 기술 (binary translation)[1]을 이용한 방법을 제안한다. 동적 코드변환 기술은 프로그램의 런-타임(run-time)에 프로그램에 코드를 변환하여 특정한 동작을 할 수 있는 기능을 제공한다. 동적 코드변환 기술은 실행 중에 동적으로 변환을 하기 때문에 프로그램 소스코드를 변경하지 않아도 되어 유연한 환경을 제공한다. 또한 변환 자체의 오버헤드가 적어 성능 좋은 소프트웨어 트랜잭션 메모리 구현에 적합하다.

* 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음 (NIPA-2010-(C1090-1021-0008))

2. 관련연구

2.1. 병렬프로그래밍

퍼스널 컴퓨터가 대중에게 널리 보급되는 시기부터 일반 데스크톱 환경은 싱글 쓰레드를 기반으로 하는 응용 프로그램이 주류를 이루었다. 이미 병렬구조를 가진 슈퍼컴퓨터로 인해 수 십 년 전부터 병렬프로그래밍에 대한 연구가 있었지만, 일반적인 응용 프로그램 개발자는 병렬 프로그래밍 기법에 익숙하지 않다. 이와 관련해 최근 데스크톱 환경의 응용프로그램 개발에 널리 사용되는 자바, C#, 루비와 같은 높은 수준의 언어는 프로그래머와 하드웨어를 분리시켜 좀 더 쉬운 프로그래밍 환경을 제공해 왔다. 프로그래밍 언어가 이러한 방향으로 발전함에 따라 많은 개발자들이 프로세서의 발전과 변화에는 무관심 했다. 현재의 프로세서는 에너지 효율 및 발열의 문제와 프로세서의 싱글 스레드에서 찾을 수 있는 소위 명령어 수준 병렬성의 한계 때문에 더 이상 클럭을 높이기는 힘들다. 대신 그 대안으로 코어 자체의 수를 늘리는 병렬 프로세서가 주목받아 이제는 대부분의 프로세서가 멀티코어로 바뀌었다. 이처럼 프로그램 실행에 중요한 역할을 하는 프로세서가 단순히 속도보다는 병렬성(Parallelism)을 증대하는 방향으로 발전하고 있기 때문에 최신 병렬프로세서의 성능을 제대로 활용하기 위해 일반 응용 프로그램에서도 병렬프로그래밍의 중요성이 부각되고 있다.

2.2. 트랜잭션 메모리

병렬프로그래밍에서 가장 어려운 것 중 하나는 바로 공유 자원 관리이다[5][6]. 일반적으로 공유 자원 관리에는 원리가 간단한 락-기반의 뮤텍스나 세마포어를 사용한다. 하지만 이를 잘못 사용해 데이터 일관성이 지켜지지 않으면 프로그램은 심각한 오류를 일으킬 수 있다. 병렬 프로그래밍은 프로그램의 동작과정을 예측하기 힘들기 때문에 이러한 버그를 다시 유발 시켜 디버깅하기 어렵다는 점에서 개발 과정에 문제점이 많다. 뿐만 아니라 락-기반의 동기화 기법은 오버헤드가 크므로 성능이 매우 떨어진다.

이런 문제점에 대한 대표적인 대안으로 소프트웨어 트랜잭션 메모리를 들 수 있다. 소프트웨어 트랜잭션 메모리는 데이터베이스(DB)의 트랜잭션 기법과 유사한 방식이다. 데이터베이스의 트랜잭션은 원자적으로 데이터를 처리하는 기능 단위를 말한다. 이와 유사하게 병렬 프로그래밍에서 “원자적(Atomic)”의 의미는 실행 도중 다른 쓰레드에 의해 방해 받지 않는 기능 단위를 말한다. 예를 들어 동기화 기법 중 하나인 임계 영역(Critical Section)은 락을 이용해 인위적으로 어떤 작업을 "원자적"으로 수행하는 방법이다. 트랜잭션은 임계 영역과 유사하게 작업을 원자적으로 수행하지만 락을 이용하지 않는 방법이다.

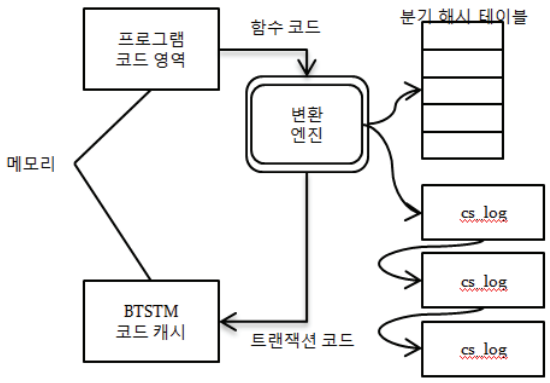
소프트웨어 트랜잭션 메모리는 이와 같은 데이터베이스 트랜잭션의 특징을 병렬 프로그래밍에 적용하여 소프트웨어 적으로 구현한 것이다. 다만 차이가 있다면 데이터베이스의 트랜잭션은 쿼리 작업의 실패와 같은 비정상 동작인 경우에 전체 작업이 복구되지만 소프트웨어 트랜잭션 메모리에서는 작업 수행 도중 다른 쓰레드에서 동일한 트랜잭션에 접근할 경우 전체 작업이 복구된다.

소프트웨어 트랜잭션 메모리는 락-프리 동기화 기법이기 때문에 동기화시 가장 큰 문제가 되는 데드-락, 우선순위 역전 등의 문제를 발생시키지 않는다[2]. 게다가 소프트웨어 트랜잭션 메모리의 기능 모듈들은 락의 획득과 해제에 대한 문제를 신경 쓰지 않아도 되기 때문에 모듈들의 융합이 락-기반 동기화에 비해 쉽다. 또한 소프트웨어 트랜잭션 메모리는 뛰어난 예외 안정성을 가지고 있다. 락-기반 동기화 방법에서는 예외 안정성을 유지하기 어렵지만 소프트웨어 트랜잭션에서는 작업이 성공하지 못하면 작업 수행 이전 상태로 복구되기 때문에 별다른 처리 없이 예외 안정성을 유지할 수 있다.

2.3. 동적 코드변환 기술(Binary Translation)

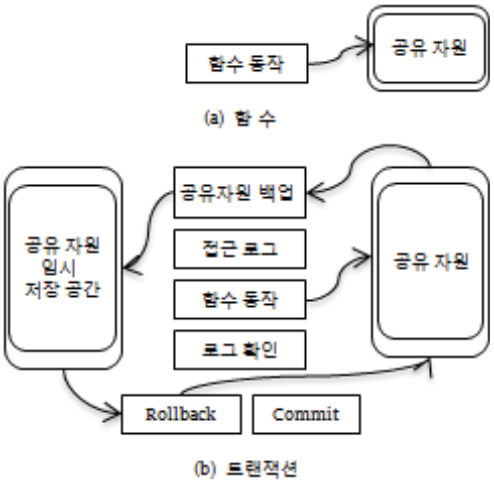
동적 코드변환 기술은 코드를 변환하여 대상 아키텍처의 명령 집합(Instruction Set)을 또 다른 명령 집합으로 에뮬레이션 하는 방법이다. 동적 코드변환 기술은 프로그램이 수행되는 베이직 블록(Basic block)의 순서대로 코드를 변환하여 베이직 블록 캐시라고 하는 다른 메모리 영역에 저장하고 실행하는 과정이다. 변환은 계속해서 반복적으로 일어나는 것이 아니라 변환되지 않은 베이직 블록을 만나거나 분기 명령(Branch Instruction)등을 만났을 때에 한해 이루어진다. 이처럼 한번 변환된 코드가 계속적으로 수행되므로 실제 프로그램의 성능에 미치는 영향은 매우 적다[1]. 뿐만 아니라 변환 과정에서 코드에 특정한 일을 해 줄 수 있기 때문에 동적 분석(Dynamic Analysis)이나 동적 최적화(Dynamic Optimization)등에도 널리 활용 되고 있는 기술이다.

3. BTSTM(Binary Translation STM)



(그림 1) BTSTM 구조도

앞에서 언급한 바와 같이 락-기반의 동기화 기법은 원천적으로 여러 가지 문제점을 가지고 있다. 소프트웨어 트랜잭션 메모리는 이러한 단점을 가진 락 기반의 동기화



(그림 2) 함수와 변환 후의 트랜잭션 비교

기법에 대한 대안으로 제시되는 락-프리 동기화 기법이다. 본 논문에서는 소프트웨어 트랜잭션 메모리를 동적 코드변환 기술로 구현 하는 BTSTM (Binary Translation Software Transactional Memory)을 제안한다.

BTSTM은 그림1과 같이 일반 함수 코드를 트랜잭션 코드로 변환하기 위한 변환 엔진과 함수 호출과 반환(return), 간접 분기(indirect branch)를 위한 분기 해시 테이블 그리고 각 임계 영역의 접근 로그 정보를 저장할 연결 리스트(linked-list)로 구성 되어 있다.

분기 해시 테이블은 코드 변환후의 간접 분기 혹은 반환 명령을 처리하기 위해 원래 코드의 주소와 변환 후 코드 캐시의 주소에 대한 매핑(mapping)정보를 저장하는 테이블이다. 그리고 여러 트랜잭션에 관한 처리를 위해 로그 정보를 위한 자료구조를 연결리스트로 유지하여 각 트랜잭션의 로그 정보를 관리 한다.

변환 엔진(translation engine)은 동적 코드변환 기술의 핵심 부분이다. 변환 엔진은 기본적으로 BTSTM의 동작을 위해 코드 변환을 수행한다. 분기(branch), 간접 분기, 함수 호출 및 반환 등 순차적으로 변환 불가능한 상황에서 적절히 베이직 블록을 나누어 원할 하게 코드 변환을 할 수 있도록 한다. 또한 변환 엔진은 소프트웨어 트랜잭션 메모리를 위해 일반 함수의 동작을 단일 명령(Instruction)단위로 분석하여 트랜잭션으로 변환 한다.

$$function(shared)$$
$$shared \leftarrow shared + 1$$

위와 같은 공유 자원을 접근하는 간단한 함수를 트랜잭션으로 변환하는 과정을 생각해 보자. 이 함수는 그림2(a) 같은 일반적인 함수 이다. 변환 엔진은 이러한 함수를 다음과 같은 동작을 추가하여 함수를 트랜잭션으로 변환한다.

1. 트랜잭션이 실패 했을 경우에 롤백(Roll-back)을 처리 하기위해 공유 자원을 백업한다.
2. 스레드가 해당 트랜잭션에 접근 할 때 마다 접근 로그를 저장 한다.
3. 원래 함수의 동작과정을 처리 한다.
4. 모든 작업이 끝나면 접근 로그 정보를 확인한다.
5. 접근 로그 정보가 변경 되었으면 공유 자원을 롤백하고 트랜잭션을 취소(Abort)하여 다시 시작 한다.
6. 접근정보가 변경 되지 않았을 경우 해당 공유 자원을 커밋하고 트랜잭션을 종료 한다.

그림2(a)와 같은 일반적인 함수에 이 과정을 적용하면 그림2(b)와 같은 트랜잭션으로 변환된다. 앞서 소개한 함수에 실질적으로 이 과정을 적용하여 트랜잭션으로 변환한 결과는 다음과 같다.

$$transaction(shared)$$
$$temp \leftarrow shared$$
$$write\_access\_log(current)$$
$$shared \leftarrow shared + 1$$
$$if\ check\_access\_log(current) = no\ access$$
$$then\ commit()$$
$$else\ shared \leftarrow temp \quad \#roll\_back$$
$$abort()$$

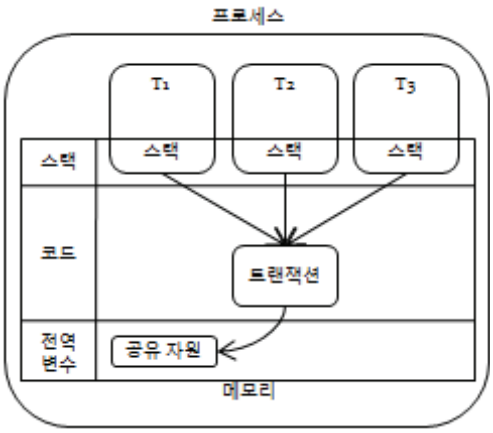
동적 코드변환 기술에서 실제 모든 변환은 머신코드 레벨에서 명령 단위로 이루어진다. 그러므로 정확한 변환 동작을 설명하기 위해선 머신코드와 1:1로 대응되는 어셈블리 코드를 사용해야한다. 그러나 직관적으로 어떠한 동작을 통해 일반 함수를 트랜잭션으로 변환하는지 알기 위해 의사 코드(Pseudo code)를 이용하였다.

4. 평가

	락-기반 동기화 기법	Non-Block STM	BTSTM
락-기반	O	X	X
데드-락	O	X	X
우선순위역전	O	X	X
성 능	느림	빠름	빠름
코드 수정	필요	필요	불필요
API 제공	X	제공	불필요

(그림 3) 공유자원 동기화 기법 비교

본 논문에서는 제안 기법을 다른 동기화 기법과 특징적인 차이의 비교를 통해 평가하고자 한다. BTSTM은 알아본 바와 같이 락-프리의 기법이기 때문에 데드-락, 우선순위역전과 같이 락-기반의 동기화 기법들이 가지는 문제



(그림 4) 트랜잭션의 지역성(Locality)

를 원천적으로 해결 할 수 있다. 또한 BTSTM은 동적 코드변환 기술의 여러 가지 특징과 장점을 이용하여 기존 비 차폐(Non-Blocking) 소프트웨어 트랜잭션 메모리와 비교해 유연하고 성능 좋은 소프트웨어 트랜잭션 메모리를 구현 할 수 있다.

동적 코드변환 기술은 런-타임(Run-time)에 코드 변환이 가능하기 때문에 이미 작성한 함수를 수정하지 않고 간단히 트랜잭션으로 사용 할 수 있다.

```
foo(shared)
    #do something with shared resources
```

위와 같은 공유 자원에 접근하는 함수를 가정하자. 기존의 락-기반 동기화 기법에서는 다음과 같이 공유 자원을 사용하는 부분을 임계 영역(Critical Section)으로 정의하고 항상 동작의 직전에 락을 얻고 동작이 끝난 후 락을 해제 해주어야 한다.

```
atomic_foo(shared)
    acquire_lock()
    foo(shared)        #critical section
    release_lock()
```

이 과정을 생략할 경우 경쟁 조건이 발생해 프로그램이 정확히 동작 할 것이라고 보장 할 수 없다. 반면에 BTSTM에서는 원자적(Atomic)으로 처리해야 하는 작업 즉, 트랜잭션에 대한 별도의 처리가 매우 간단하다. BTSTM은 락-기반 동기화 기법과 달리 임계 영역 정의를 위한 별도의 함수가 필요하지 않다.

```
transaction (foo, shared)
```

위와 같이 BTSTM에서 제공하는 transaction함수를 통해 처리할 함수를 호출함으로써 간단하게 트랜잭션을 수행 할 수 있다.

또한 동적 코드변환 기술은 최초에 코드를 변환하고 나면 그 이후에는 변환된 코드를 계속하여 실행 하므로 지역성이 높은 코드에 대하여 좋은 성능을 낸다. 그림4와 같이 병렬프로그래밍에서 공유 자원을 접근하는 코드는 각각의 쓰레드에 의해 빈번히 사용 되어 지역성(Locality)이 매우 높다. 일단 함수가 트랜잭션으로 변환되면 이후에는 변환 없이 모든 쓰레드가 트랜잭션을 바로 호출 하므로 BTSTM에서 함수를 트랜잭션으로 바꾸는데 드는 비용이 전체 프로그램 수행비용에서 차지하는 비율은 매우 낮다. 이로 인해 BTSTM은 기존에 이미 구현된 많은 함수를 트랜잭션으로 활용 할 수 있으며 특정 API를 이용하여 트랜잭션을 생성하는 비 차폐(Non-Blocking) 소프트웨어 트랜잭션 메모리와 성능 면에서도 큰 차이가 없다.

5. 결론 및 향후 연구

본 논문에서는 동적 코드변환 기술을 이용한 소프트웨어 트랜잭션 메모리기법을 제안하였다. 프로세서의 발전과 함께 중요해진 병렬프로그래밍에서 공유자원의 경쟁 조건 문제를 해결하기 위한 방법으로 지역성 코드 성능 향상의 특징을 가진 동적 코드 변환기술을 통해 간단하고 강력한 소프트웨어 트랜잭션 메모리 기법을 구현하는 방법을 제안 하고 기존의 구현 방법과 비교 평가 하였다.

참고문헌

[1] R. Sites, A. Cherno, M. Kirk, M. Marks, and S. Robinson, "Binary translation," Communications of the ACM, 36(2), 69-81 (1993).

[2] N. Shavit and D. Touitou. Software transactional memory. Distributed Computing, 10(2), 99-116 (1997).

[3] Herb Sutter and James Larus," Software and the concurrency revolution," ACM Queue, 3(7), 54-62 (2005).

[4] Jones, S. P., Beautiful Concurrency, in Beautiful Code, Wilson, G., O'Reilly, (2007).

[5] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. Cao-Minh and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," In proc. of the 11th ACM SIGPLAN symposium on Principles and practive of parallel programming (2006).

[6] H. Sutter, "The Free Lunch Is Over: A Fundamental TurnToward Concurrency in Software," Dr. Dobb's Journal, 30(3), (2005).