공학석사 학위논문

사전 스케줄링을 통한 이더리움 트랜잭션 동시 수행

Speculative Transaction Scheduling for Adding Concurrency to Ethereum

2021년 2월

서울대학교 대학원

전기정보공학부

구 연 재

사전 스케줄링을 통한
이더리움 트랜잭션 동시 수행

지도 교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함

2021년 2월

서울대학교 대학원

전기정보공학부

구 연 재

구연재의 공학석사 학위논문을 인준함

2021년 2월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 엄 현 상 (인)

초 록

이더리움[1]은 대표적인 블록체인 플랫폼으로, 암호화폐로부터 시작된 블록체인 기술에 대한 관심을 바탕으로 최근까지도 사용량을 꾸준히 유지하고 있다. 특히 유니스왑[2]과 같은 탈중앙화 금융 서비스들이 인기를 끌며 일일 토큰 전송량은 2020년들어 두 배 가까이 증가하기도 했다[3]. 하지만 점점 늘어나는 트랜잭션 수에도 불구하고, 현재 이더리움은 모든 트랜잭션을 순차적으로 수행하고 있기에 트랜잭션의 처리량을 높이는 데에 한계가 있다. 실제로 이로 인해 사용자가 많은 서비스가 등장할 때마다 트랜잭션 처리량은 그대로인데 트랜잭션의 수가 증가하면서 수수료가 매우 높아지는 현상이 발생하고 있다[4]. 이러한 문제점을 해결하기 위한 방법 중 하나로 이더리움 상의 트랜잭션들을 동시 수행하기 위한 연구들이 제안되어왔다. 하지만 이러한 동시 수행은 트랜잭션간의 충돌을 유발할 수 있으며, 충돌이 자주 발생할 경우 오히려 성능이 저하될 수 있다.

따라서 본 논문에서는 충돌이 최대한 적게 발생하도록 스케줄링하는 것에 초점을 맞춘 사전 스케줄링을 통한 동시 수행 기법을 제안한다. 트랜잭션에 명시된 정보로 쉽게 예측 가능한 명시적 충돌(explicit conflict)뿐 아니라 직접 수행해보아야 확인이 가능한 내재적 충돌(implicit conflict)까지도 과거 충돌 정보를 저장하는 프로파일러를 통해 예측함으로써, 4-thread 시뮬레이션에서 순차 수행 대비 전체 트랜잭션 수행 속도를 약 3.1배, 기존의 그리디 스케줄링 방식의 동시 수행 대비 약 28.2% 향상시켰다.

주요어 : 이더리움, 블록체인, 스마트 컨트랙트, 트랜잭션, 동시수행, 스케줄링
학 번 : 2019-24567

목 차

표 목차

그림 목차

제 1 장 서 론

최근 몇년간, 암호화폐로부터 시작된 블록체인 기술에 대한 관심을 바탕으로 비트코인, 이더리움을 비롯해 다양한 블록체인 플랫폼들이 인기를 끌며 그 사용량이 꾸준히 증가해왔다. 특히 이더리움은 단순히 돈을 송금하는 트랜잭션뿐 아니라 스마트 컨트랙트라는 일종의 프로그램을 수행하는 트랜잭션을 제공한다는 특징을 가지고 있는데[1], 이를 통해 만들어지는 탈중앙화 어플리케이션(DApp) 서비스가 다양해져가며 일반적인 트랜잭션보다 스마트 컨트랙트를 호출하는 트랜잭션이 더욱 많아지는 추세이다[5]. 특히 유니스왑[2]과 같은 금융 서비스들이 인기를 끌며 일일 토큰 전송량이 2020년들어 두 배 가까이 증가하기도 했다[3]. 이러한 트랜잭션 발생 양상은 블록체인이 좀 더 생활의 다양한 서비스에 접목되며 사용량이 증가해감과 동시에 처리하는 명령 또한 점점 복잡해져감을 시사한다. 하지만 이렇게 점점 많아지고, 복잡해져가는 트랜잭션에도 불구하고, 현재 이더리움은 모든 트랜잭션을 순차적으로 수행하고 있기에 트랜잭션의 처리량을 높이는 데에 한계가 있다. 실제로 이로 인해 사용자가 많은 서비스가 등장할 때마다 트랜잭션 처리량은 그대로인데 트랜잭션의 수가 증가하면서 수수료가 매우 높아지는 현상까지 발생하고 있다[4]. 따라서 이러한 순차 수행 디자인은 점점 많아지고 복잡해져가는 트랜잭션의 발생 양상에 비해 현대의 멀티코어 아키텍처 리소스들을 충분히 활용하지 못하고 있다는 비효율성을 안고 있다.

최근 이러한 한계를 해결하기 위해 이더리움 상의 트랜잭션을 동시 수행하기 위한 기법들이 제안되어왔다[6]. 하지만 트랜잭션을 단순히 그리디(greedy)한 방식의 스케줄링을 통해 동시 수행할 경우 트랜잭션 간 충돌을 사전에 막을 수 없으며, 이로 인해 충돌이 자주 발생할 경우 오히려 성능이 저하될 수 있다. 3장에서 다룰 이더리움 워크로드 분석 결과, 실제로 이더리움의 한 블록당 평균 충돌률은 64.5% 정도로 높은 편이다.

본 논문에서는 이러한 이더리움 트랜잭션 충돌 양상에 주목하여, 동시 수행 시 충돌이 최대한 적게 발생하도록 스케줄링하는 것에 초점을 맞춘 사전 스케줄링 기법을 제안한다. 예를들어, 기존 그리디 스케줄링 방식의 동시 수행이라면 그림 1(b)과 같이 충돌이 발생하는 상황에서, 사전에 이러한 충돌을

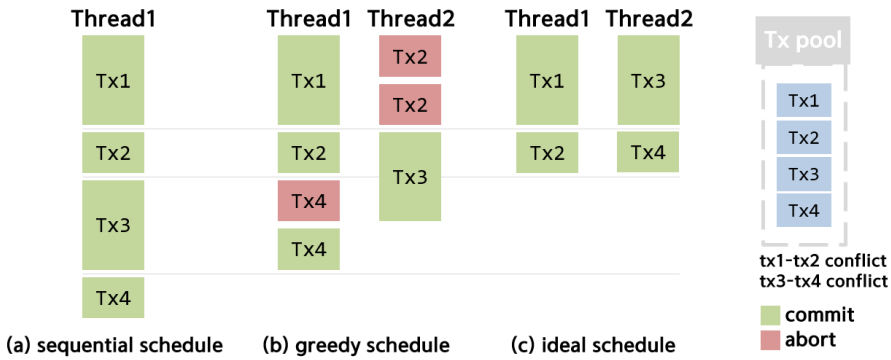


그림 1 스케줄링 방식에 따른 트랜잭션의 수행

미리 예측하고 스케줄링하여 그림 1(c)와 같이 충돌이 발생하지 않도록 하는 것이다. 이러한 충돌의 사전 예측을 위하여, 본 논문에서는 충돌의 종류를 트랜잭션에 명시된 정보로 쉽게 예측 가능한 명시적 충돌(explicit conflict)과 직접 수행해보아야 확인이 가능한 내재적 충돌(implicit conflict)로 나누어 정의하였고, 내재적 충돌의 경우 스케줄링 단계에서는 충돌 여부는 확인할 수 없으므로 과거 충돌 기록을 기반으로 한 프로파일러를 통해 예측하였다. 따라서 본 연구의 컨트리뷰션은 다음과 같다.

- 이더리움 실제 트랜잭션 간 충돌 양상 분석
- 프로파일링 기반의 사전 스케줄링 기법 제안 및 실제 이더리움 워크로드에 대한 시뮬레이션

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 이더리움 트랜잭션 동시 수행을 위해 알아야 할 이더리움의 기본 개념 및 트랜잭션 간의 충돌을 정의한다. 3장에서는 실제 이더리움 워크로드를 대상으로 충돌의 양상을 분석하였으며, 이후 4장에서는 이러한 충돌 양상을 고려하여 프로파일링 기반 사전 스케줄링을 제안한다. 5장에서는 이에 대한 시뮬레이션 방법 및 결과를 논의하고, 끝으로 6장에서는 관련 연구를, 7장에서는 결론과 향후 연구 방향에 대해 소개할 것이다.

제 2 장 배경지식

제 1 절 이더리움 기본 개념

계정(Account)

이더리움 상의 계정[7]은 외부 소유 계정(EOA, External Owned Account)과 컨트랙트 계정(CA, Contract Account) 두 가지 종류로 나눌 수 있다. 외부 소유 계정은 이더리움 내에서 지갑 역할을 하는 계정으로, 이더 잔고와 논스(Nonce) 값을 지니고있다. 논스값이란 해당 계정이 몇 번의 트랜잭션을 발생시켰는지를 뜻한다. 마찬가지로 컨트랙트 계정 또한 이더 잔고와 논스값을 지니고 있으며, 외부 소유 계정과는 다르게 별도로 컨트랙트 코드와 스토리지 공간을 저장하고 있다. 그림 2는 컨트랙트 코드의 예시로, 이러한 코드가 이더리움 바이트코드로 변환되어 컨트랙트 계정의 코드 공간에 저장된다. line 3에 선언된 변수는 컨트랙트 계정의 스토리지 공간에 저장되는 예시로, 이러한 변수나 함수는 트랜잭션을 통해 호출하고 접근할 수 있으며 이더리움 가상머신(EVM, Ethereum Virtual Machine) 위에서 수행된다. 이때, 스마트 컨트랙트는 튜링 완전(turing-complete)한 언어로 작성되므로 그림 2의 line 7-9와 같이, 직접 함수를 수행해보기 전에는 어떤 변수의 값에 접근할 지 미리 알 수 없다.

```
1   contract Token {
2     // 컨트랙트의 스토리지에 저장되는 변수
3     mapping(address => uint256) balances;
4
5     // 컨트랙트의 스토리지에 접근하는 함수
6     function SetBalanceOf(address addr) public {
7       if (balances[addr] > 10) {    // read
8         balances[addr] -= 1; // read, write
9       }
10    }
11  }
```

그림 2 컨트랙트 코드 예시

트랜잭션(Transaction)

이더리움에서 발생하는 트랜잭션은 대표적으로 수신자(receiver), 전송하는 이더(ether), 데이터(data) 필드를 갖고 있다. 단순히 이더를 송금하는 트랜잭션의 경우 데이터 필드는 채워져있지 않으며, 컨트랙트의 생성 또는 컨트랙트 함수를 호출하는 트랜잭션의 경우 이에 필요한 정보들이 데이터 필드에 담기게 된다. 본 논문에서는 이렇듯 트랜잭션에서 명시적으로 저장되는 필드를 통해 유추할 수 있는 데이터 접근 정보를 **명시적 정보**(explicit feature), 트랜잭션에 명시적으로 저장되지 않으며 직접 수행해보아야 알 수 있는 데이터 접근 정보, 가령 트랜잭션 수행 도중 접근하는 스토리지와 같은 정보들을 **내재적 정보**(implicit feature)라 정의한다[그림 4(b)].

채굴 및 검증 단계에서의 트랜잭션 수행

이더리움에서 트랜잭션의 수행은 채굴(mining)단계와 검증(validation)단계에서 이루어진다. 먼저, 처음 발생한 트랜잭션들은 블록에 담기기 위해 이더리움 네트워크 내의 각 노드들에게 전파된다. 채굴자 역할을 하는 노드들은 이러한 트랜잭션들을 수행하며 변화되는 상태를 블록에 기록하고, 완성한 블록을 주변 노드에 전파시킨다. 이렇게 블록을 전파받은 노드들은 블록 내의 트랜잭션들을 다시 한 번 수행해보며 블록에 기록되어있는 최종 상태가 올바른 상태인지 검증하는 과정을 거친다.

트랜잭션 동시 수행 및 사전 스케줄링

이때, 본 논문에서 의미하는 트랜잭션 동시 수행과 이를 위한 사전 스케줄링이란, 채굴 단계에서의 동시 수행과 이를 위한 스케줄링을 의미한다. 기존 이더리움의 채굴에서는 트랜잭션이 순차적으로 하나씩 수행되기에 트랜잭션 간 충돌을 고려하지 않고 그리디한 방식으로 스케줄링 했지만, 여러 트랜잭션을 동시에 수행하는 상황을 가정할 경우 트랜잭션간 충돌이 일어날 수 있고 이는 오히려 성능 저하를 유발할 수 있으므로 이를 고려하여 스케줄링 하자는 것이다. 이후 검증 단계에서의 트랜잭션 동시 수행을 다루기 위해서는, 채굴 단계에서 블록에 트랜잭션을 담을 때, 동시 수행된 트랜잭션들의 직렬 순서가 담겨야 할 것이다. 또한 검증자들은 이러한 직렬 스케줄과 동등한 병렬 스케줄을 만들어내는 과정이 필요할 것이다.

제 2 절 이더리움 트랜잭션 충돌

전통적인 데이터베이스 분야와 마찬가지로 이더리움에서도 여러 트랜잭션을 동시에 수행하는 경우 같은 데이터에 동시에 접근하게되는 충돌(conflict)이 발생할 수 있으며, 이로 인해 일관성(consistency)이 유지되지 않을 수 있다. 예를 들어 그림 4(b)의 두 트랜잭션 Tx1과 Tx2를 동시에 수행할 경우, 같은 계정 0xaa의 잔고(balance)와 논스(nonce) 값에 동시에 접근하게 되므로 충돌이 발생하며, 이로 인해 일관성이 깨지게 될 것이다.

이더리움 상에서 동시에 같은 값에 접근하게 되는 문제의 대상은 비단 잔고와 논스뿐만이 아니다. 그림 4(b)의 Tx2와 Tx3을 동시에 수행할 경우, 컨트랙트 계정 0xcc의 스토리지(storage) 값 역시 동시에 읽고, 쓰여지므로 동시 수행 시 이러한 일관성 유지에 주의해야 한다.

이때, 트랜잭션 상에는 명시적으로 발신자, 수신자, 전송하는 이더 값 등의 정보가 포함되어 있으므로 Tx1과 Tx2간의 충돌은 트랜잭션을 수행해보지 않더라도 명시적 정보를 통해 자명하게 알 수 있다[그림 3]. 반면, Tx2와 Tx3 간의 충돌과 같이 가상머신 상에서 수행하며 발견되는 충돌의 경우, 스마트 컨트랙트가 튜링 완전 언어로 작성되기에 수행 전에는 미리 알 수 없다. 본 논문에서는 이렇듯 수행해보지 않아도 트랜잭션 상의 정보로 파악할 수 있는 충돌을 **명시적 충돌**(explicit conflict), 수행 전에는 미리 파악할 수 없는 충돌을 **내재적 충돌**(implicit conflict)이라 정의한다[그림 4(c)].

```
1   def is_explicit_conflict_pair_tx(tx1, tx2):
2     # check nonce conflict
3     if tx1['from'] == tx2['from']:
4       return True
5
6     # check balance conflict
7     if tx1['balance'] > 0 and tx2['balance'] > 0:
8       if set([tx1['from'], tx1['to']]) | set([tx2['from'], tx2['to']]):
9         return True
10
11    return False
```

그림 3 명시적 충돌 검사 코드

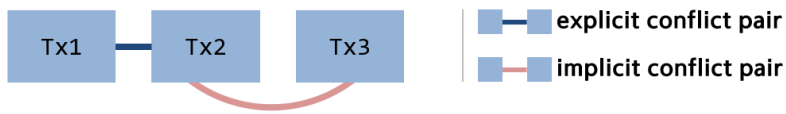
0xaa (EOA)	0xbb (EOA)	0xcc (CA)
Nonce : 1 Balance: 1 Ether Storage: null Code : null	Nonce : 0 Balance: 1 Ether Storage: null Code : null	Nonce : 0 Balance: 1 Ether Storage: [..] Code : [..]

Tx1	Tx2	Tx3
From : 0xaa To : 0xbb Value: 0.01 Ether Data : null	From : 0xaa To : 0xcc Value: 0.1 Ether Data : Call foo()	From : 0xbb To : 0xcc Value: 0 Ether Data : Call foo()

(a) 예시 트랜잭션 및 계정

	explicit feature	implicit feature
Tx1	read/write 0xaa(nonce,balance) read/write 0xbb(balance)	-
Tx2	read/write 0xaa(nonce,balance) read/write 0xcc(balance)	read/write 0xcc(storage)
Tx3	read/write 0xbb(nonce)	read/write 0xcc(storage)

(b) 예시 트랜잭션에 대한 명시적 정보와 내재적 정보



(c) 트랜잭션 간 명시적 충돌 및 내재적 충돌 관계

그림 4 트랜잭션의 명시적·내재적 충돌 예시

제 3 장 이더리움 워크로드 분석

사전 스케줄링을 통한 동시 수행을 제안하기에 앞서, 사전 스케줄링의 필요성을 설명하고 실제 이더리움 워크로드에 더 적합한 스케줄러를 디자인하기 위하여 먼저 이더리움 상에서 발생하는 트랜잭션들 간 실제 충돌 양상을 분석해보았다.

제 1 절 데이터 수집

분석에는 이더리움 발표일인 2015년 7월 30일부터 2020년 9월 10일까지의 10,833,600 블록 데이터를 100 블록 단위로 샘플링하여 사용했으며, 따라서 총 108,336 블록(8,322,521 트랜잭션)의 데이터를 분석하였다.

충돌 분석에는 트랜잭션에 저장되는 명시적인 정보(발신자, 수신자, 전송 이더, 소모 가스 등)뿐 아니라, 트랜잭션을 수행하며 어떤 스토리지에 접근하는지, 내부 트랜잭션으로 인해 명시적인 발신자, 수신자 외 다른 계정의 데이터에 접근하지는 않았는지 등의 내재적 정보가 필요했다. 블록체인 데이터 상에는 이러한 수행의 결과만이 담기게 되므로 내재적 정보를 기록하기 위해서는 이더리움 클라이언트에 구현된 명령어 핸들러 함수에서 로그를 남기도록 수정해야 했다. 수정한 명령어 핸들러 함수의 목록은 선행연구 [9]를 참조하여 컨트랙트의 생성·호출·삭제 함수(CREATE, CALL, CALL-CODE, DELEGATECALL, SELFDESTRUCT), 잔고 확인 및 송금 함수(BALANCEOF, TRANSFER), 스토리지 읽기 및 쓰기 함수(SLOAD, SSTORE)를 수정하였다. 이


```
{
  "blocknumber": 68400,
  "from": "0xb053b1d48cc216942e431ca955a29476d954efdd",
  "to": "0x7157d2d644cf4006edbad877f532abff068da8de",
  "nonce": 1,
  "balance": "15462600000000000",
  "operations": ["TRANS 0xb053b1d48cc216942e431ca955a29476d954efdd
  0x7157d2d644cf4006edbad877f532abff068da8de"],
  "return": "null",
  "type": 0,
  "gas": 21000,
  "txid": 3

}
```


그림 5 한 트랜잭션에 대한 로그 예시

를 통해, 그림 5와 같이 트랜잭션의 명시적 정보뿐 아니라, 이더리움 가상머신 상에서 수행한 명령어의 종류와 인자에 대한 내재적 정보인 operation 항목도 기록할 수 있었다.

제 2 절 분석 결과

명시적 충돌 및 내재적 충돌의 비율

이더리움 상에서 발생하는 트랜잭션들 간 실제 충돌 양상을 분석하기위해, 먼저 앞서 정의한 명시적 충돌 및 내재적 충돌이 한 블록 내에서 각각 어느 정도 비율로 발생하는 지 알아보았다. 한 블록 내의 모든 트랜잭션이 동시에 수행된다고 가정했을 때, 동일한 값(잔고, 논스, 스토리지 값)에 접근하는 다른 트랜잭션이 하나라도 존재한다면 충돌 트랜잭션으로 간주하였다. 블록의 충돌률은 블록 전체 트랜잭션 수 대비 충돌 트랜잭션의 수를 뜻한다.

분석 결과, 그림 6에서 볼 수 있듯 명시적 충돌의 비율은 세번째 구간(2,000,000~3,000,000 블록)에서 78.6%로 매우 높은 양상을 나타낸 뒤, 이후 급격히 감소하여 현재까지 약 50% 수준을 유지하고 있는 데에 반해, 내재적 충돌의 비율은 오히려 두번째 구간(1,000,000~2,000,000)에서 0.1%로 최저점을 기록한 이후 꾸준히 증가 추세로, 가장 최근 구간에서는 15.3%까지 도달한 것을 확인할 수 있었다. 초반에 명시적 충돌이 매우 높은 비중을 차지한 이유는, 이더리움이 보편화되지 않아 트랜잭션이 일부 사용자에 의해서만 발생하였기 때문에 한 블록 내의 트랜잭션이 동일한 발신자인 경우가 많기 때문이다. 하지만 점차 다양한 사용자로부터 트랜잭션이 발생하게 되었고,

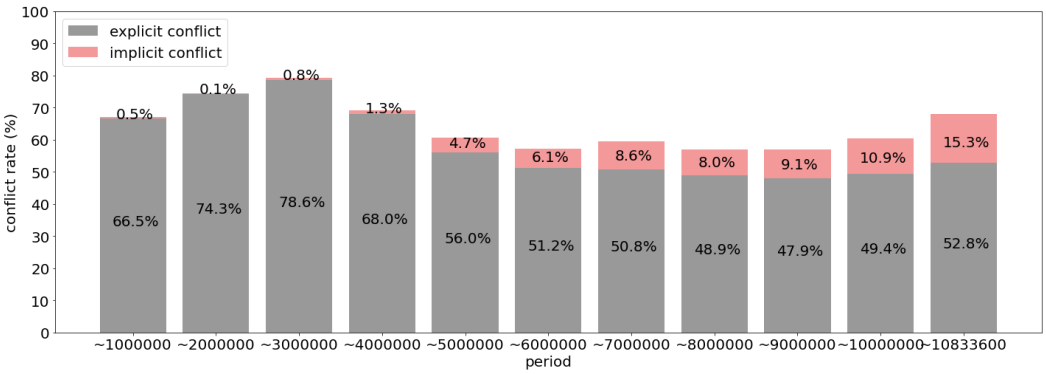


그림 6 시기별 충돌 트랜잭션 비율

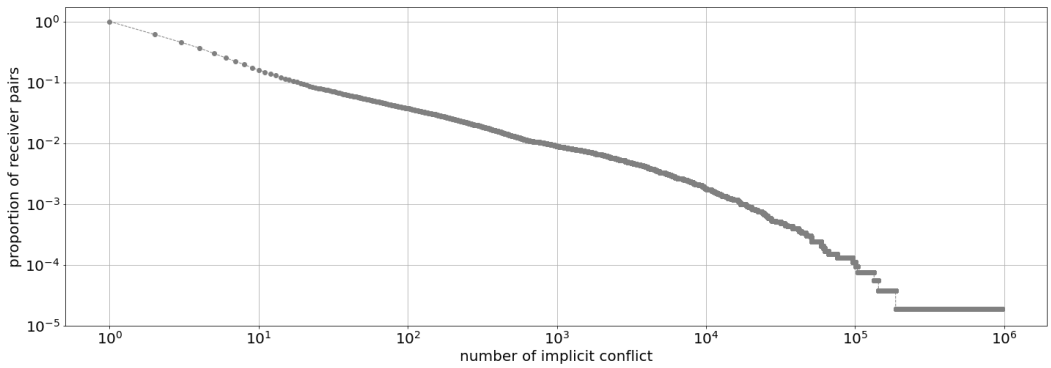


그림 7 내재적 충돌 트랜잭션을 야기하는 계좌 쌍 비율

이로 인해 오히려 한 블록에 대한 동시성은 높아져 명시적 충돌이 점점 감소하는 양상을 띤 것으로 추정된다. 반면 내재적 충돌 증가의 경우, 과거에 비해 스마트 컨트랙트를 호출하는 트랜잭션의 비율이 많아지면서[5], 명시적으로는 충돌이 없는 트랜잭션이더라도 트랜잭션의 수행 도중 같은 스토리지를 접근하거나 잔고의 이동이 발생한 것이 영향을 끼친 것으로 보인다. 특히 가장 최근 구간(10,000,000~ 10,833,600)에서 이전 구간(9,000,000~10,000,000)에 비해 10.9%에서 15.3%로 눈에 띄게 내재적 충돌 트랜잭션의 비율이 증가하였는데, 이러한 내재적 충돌 트랜잭션의 큰 증가에는 10,207,858번째 블록에서 생성된 유니스왑 컨트랙트[1]가 큰 영향을 미친 것으로 생각된다.

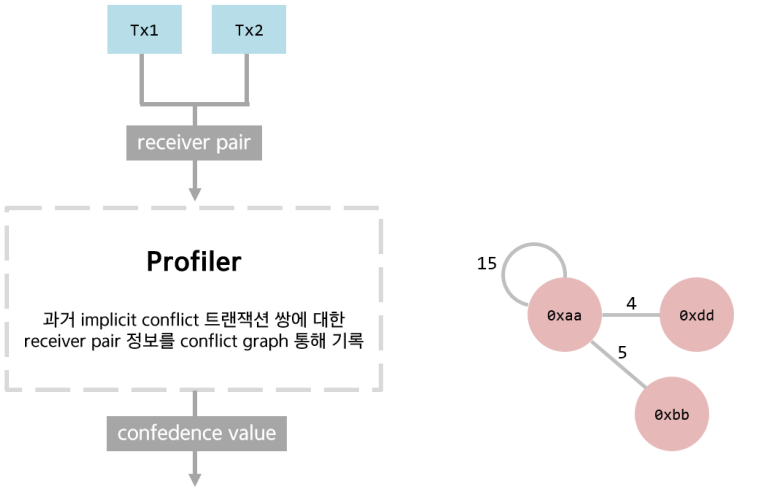
내재적 충돌 발생 양상 관찰

이렇듯 내재적 충돌이 점점 증가해가며 최근에는 전체 충돌의 22% 가량을 차지하고 있는 양상이기에, 동시성을 최대한 높이기 위해서는 이러한 내재적 충돌도 무시할 수 없다. 내재적 충돌은 트랜잭션을 직접 수행해보며 얻을 수 있는 내재적 정보를 통해서만 파악할 수 있긴 하지만, 우리는 이러한 충돌을 명시적 정보만을 이용해 스케줄링 해야했기에 내재적 충돌이 발생한 트랜잭션들의 발생 양상을 관찰해보았다.

그림 7 은 내재적 충돌이 발생한 트랜잭션 간의 수신자 쌍의 분포를 x, y 좌표를 로그스케일로 하여 나타낸 것이다. 샘플링된 108,336 블록의 충돌을 분석하는 과정에서 발견된 트랜잭션 간 내재적 충돌은 총 5,911,464 개 였으

[1] 0x7a250d5630b4cf539739df2c5dacb4c659f2488d

며, 이를 발생시킨 수신자 쌍의 수는 53,233 개였다. 그래프의 (x, y) 좌표는 x 개 이상의 내재적 충돌을 발생시킨 수신자 쌍이 전체에서 y 만큼의 비율을 차지한다는 뜻으로, 즉 (1, 1)은 1 개 이상의 내재적 충돌을 발생시킨 수신자 쌍은 53,233 개이며 (100000, 0.0001)은 100,000 개 이상의 내재적 충돌을 발생시킨 수신자 쌍이 약 5 개(53233*0.0001)임을 의미한다. 이렇게 그려진 그래프가 선형임을 통해, 내재적 충돌을 발생시키는 수신자 쌍의 분포가 멱 법칙(power law)를 따른다는 것을 알 수 있는데, 이로부터 곧 대부분의 내재적 충돌이 소수의 수신자 쌍에 의해 발생한다는 중요한 사실을 알아낼 수 있었다. 실제로, 상위 0.01%인 532 개의 수신자 쌍이 전체 내재적 충돌의 88.37% (5,224,018)를 발생시킨다. 이러한 사실을 통해 우리는 과거에 내재적 충돌이 발생한 트랜잭션들의 명시적 정보인 수신자 정보를 통해 내재적 충돌을 경험적으로 유추할 수 있을 것으로 기대했고, 이를 위해 과거 내재적 충돌의 정보를 기록하기 위한 프로파일러를 제안하였다[그림 8(a)]. 이후 4 장에서 이러한 프로파일러의 구현과, 프로파일러를 토대로 한 스케줄링 기법에 대해 다룬다.



(a) 프로파일러 구조 (b) 프로파일러의 충돌 그래프

그림 8 과거 내재적 충돌 정보를 기록하는 프로파일러 구조

제 4 장 사전 스케줄링을 통한 트랜잭션 동시수행

앞서 3장에서 분석한 것과 같이 평균적으로 한 블록 내의 64.5% 가량이 충돌 트랜잭션이라는 점은 동시 수행할 때 단순히 그리디한 방식을 취하여 스케줄링할 경우, 잦은 충돌로 인해 동시성이 떨어질 수 있다는 의미를 내포한다. 따라서 기존의 그리디한 스케줄링 방식보다는, 트랜잭션 풀에 있는 트랜잭션 간의 충돌을 예상하여, 충돌이 최대한 적게 발생하도록 스케줄링하는 것이 동시성을 높이는 데 더욱 유리할 것이다. 따라서 본 연구에서는 이러한 목적을 달성하기 위해, 사전 스케줄링을 통한 이더리움 트랜잭션 동시 수행을 제안한다. 본 연구에서 제안하는 사전 스케줄링이란 채굴자가 다음에 수행할 트랜잭션을 선택하는 시점에서, 동시 수행중인 다른 트랜잭션들과 최대한 충돌을 막을 수 있는 트랜잭션을 선택할 수 있도록 하여 기존 그리디한 스케줄링 대비 더 큰 속도 향상을 도모하는 것이다.

이를 위해서는 결국 수행중인 트랜잭션들과 다음에 수행할 수 있는 트랜잭션 후보들이 주어졌을 때, 이들 간 발생할 수 있는 충돌을 성공적으로 예측해야 한다. 아래 1절과 2절에서는 각각 명시적 충돌을 예측하는 알고리즘과, 내재적 충돌을 예측하기 위해 과거 충돌 정보를 기록하는 프로파일러를 소개한다. 3절에서는 이 두 충돌에 대한 예측을 토대로 스케줄링에 사용한 알고리즘을 다룬다.

제 1 절 명시적 충돌 예측

명시적 충돌은 트랜잭션 상에 명시된 정보들을 통해 간단히 예측 가능하다. 2장에서 언급한 명시적 충돌의 정의에 따라, 우선 동일 논스 값에 접근하는 트랜잭션, 즉 동일한 발신자를 가진 트랜잭션이라면 이는 충돌이다. 또한 서로 다른 수신자이더라도, 두 트랜잭션의 수·발신자 집합 중 최소 한 개 이상의 교집합 원소가 존재하면서, 두 트랜잭션 모두 이더의 이동이 있다면 동일 계정의 잔고 값에 대한 접근이 되므로 이는 충돌이다. 이러한 논스, 잔고 충돌 두 경우 이외에 두 트랜잭션 사이에 발생할 수 있는 다른 명시적 충돌은 존재하지 않는다[그림 3].

제 2 절 내재적 충돌 예측

3장에서 다루었듯, 대부분의 내재적 충돌은 소수의 수신자 쌍 간에 발생한다. 따라서 이러한 과거 기록을 이용한다면 트랜잭션의 명시적인 수신자 정보만으로 내재적 충돌을 예측해볼 수 있을 것이다. 이를 위해 과거 수신자 쌍 간 충돌 정보를 기록하는 **프로파일러**[그림 8]를 아래와 같이 구현하였다.

프로파일러 자료구조

프로파일러는 과거에 발생한 내재적 충돌의 수신자 쌍 정보를 기록하기 위해 충돌 그래프(conflict graph)를 가진다[그림 8(b)]. 충돌 그래프는 무방향 가중치 그래프로, 그래프의 노드는 내재적 충돌이 발생했던 트랜잭션의 수신자를, 노드 사이를 연결하는 간선은 두 수신자를 향하는 트랜잭션이 동시에 수행된 이력이 있음을, 간선의 가중치는 동시에 수행되었을 때의 충돌 발생 정도를 의미한다. 따라서 가중치의 값이 클수록 두 수신자를 향하는 트랜잭션 간 내재적 충돌이 더 많이 발생했었음을 뜻한다.

프로파일러 알고리즘

프로파일러의 충돌 그래프를 관리하기위한 함수에는 다음과 같이 `ADD`, `REDUCE`, `RESET`, `GET_CONFIDENCE` 함수가 존재한다.

`ADD` 함수의 경우 주어진 두 노드 간 간선의 가중치를 `INC_RATE` 만큼 증가시키며, 만약 간선이 존재하지 않았다면 `INC_RATE` 의 가중치를 가진 간선을 추가한다.

`REDUCE` 함수의 경우 주어진 두 노드 간 간선의 가중치를 `DEC_RATE` 만큼 감소시키며, 만약 간선이 존재하지 않았다면 `-DEC_RATE` 의 가중치를 가진 간선을 추가한다. 이 두 함수는 트랜잭션 수행 도중 내재적 충돌이 발생하거나 발생하지 않았을 때, 이러한 정보를 충돌 그래프에 반영하기위해 사용된다.

`RESET` 함수의 경우 프로파일러로 인한 공간 및 시간적 오버헤드를 최소화하기 위해 필요한 정보만 남기고 주기적으로 충돌 그래프를 초기화하는 동작을 수행한다. 가중치가 `SATURATED_WEIGHT`를 넘는 간선의 경우 가중치

를 THRESHOLD 값으로 초기화하며(SATURATED_WEIGHT >= THRESHOLD), 그렇지 못한 간선은 모두 제거한다. 이때, 프로파일러는 두 노드 간에 간선이 존재하며 그 간선의 가중치가 THRESHOLD 이상일 경우, 두 노드를 수신자로 가지는 트랜잭션 간에 내재적 충돌이 발생할 것으로 예측한다.

GET_CONFIDENCE 함수는 이러한 결과를 반환하는 함수로, 앞서 언급한 것처럼 주어진 두 노드 간 간선의 가중치가 THRESHOLD 이상인 경우 CONFLICT_OFFSET을, 미만인 경우 NONCONFLICT_OFFSET을, 간선 자체가 존재하지 않을 경우 DEFAULT_OFFSET 값을 반환한다. 이때 반환하는 값은 3절에서 정의하게 될 컨피던스 값에 더해져서 스케줄링 시 고려된다.

제 3 절 프로파일 기반 사전 스케줄링

본 논문에서는 위 1, 2절에서 제안한 명시적 충돌 및 내재적 충돌에 대한 예측을 통해 프로파일 기반 스케줄러를 구현하였다[그림 9(a)]. 스케줄러는 수행되어야 하는 트랜잭션들과 현재 수행중인 트랜잭션들 간의 컨피던스 값을 계산하여 가장 컨피던스 값이 낮은 트랜잭션을 반환한다. 이때 컨피던스 값이란 트랜잭션의 충돌 가능성을 의미한다[8]. 즉, 컨피던스 값이 가장 높은 트랜잭션은 현재 이미 수행중인 다른 트랜잭션들과 충돌이 발생할 가능성이 가장 높은 트랜잭션이며, 컨피던스 값이 가장 낮은 트랜잭션은 이미 수행중인 다른 트랜잭션들과 충돌 가능성이 가장 낮은 트랜잭션이다. 따라서 스케줄러는 트랜잭션 풀에 있는 트랜잭션들에 대해 이를 계산하고, 가장 컨피던스 값이 낮은 트랜잭션을 반환하여 유휴상태 쓰레드에 스케줄링한다.

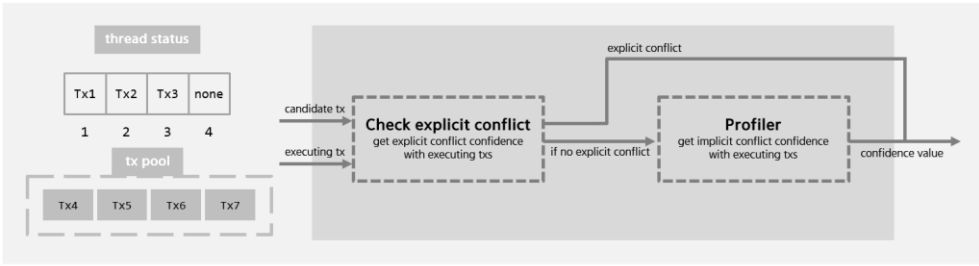
어떤 한 후보 트랜잭션의 컨피던스 값의 계산방법은 다음과 같다. 먼저 이미 수행중인 다른 트랜잭션과 하나라도 명시적 충돌이 발견되었다면 이미 해당 후보 트랜잭션은 스케줄링 되었을 때 충돌이 발생할 것임이 자명하므로 이후 다른 트랜잭션과의 충돌 검사는 따로 거치지 않은 채 컨피던스 값으로 무한대를 설정한다[그림 9(b)]. 하지만 만약 수행중인 어떤 트랜잭션과도 명시적 충돌이 없었다면, 프로파일러의 GET_CONFIDENCE 함수를 통해 수행중인 각 트랜잭션들과의 컨피던스 값을 구하고, 이를 모두 합하여 최종 컨피던스 값을 계산한다[그림 9(c)]. 이렇게 모든 후보 트랜잭션에 대해 컨피던스 값을 계산하고나면, 가장 값이 작은 후보 트랜잭션을 유휴상태인 쓰레드에

할당한다. 어떠한 한 시점에 대해 유휴상태인 쓰레드를 모두 채우고나면, 그 시점에서의 스케줄링이 완료된 것이며 이를 수행해보았을 때의 충돌 여부에 따라 프로파일러의 ADD, REDUCE 함수를 통해 충돌 그래프를 업데이트함으로써 최신 충돌 정보를 계속 다음 스케줄링에 반영할 수 있도록 한다.

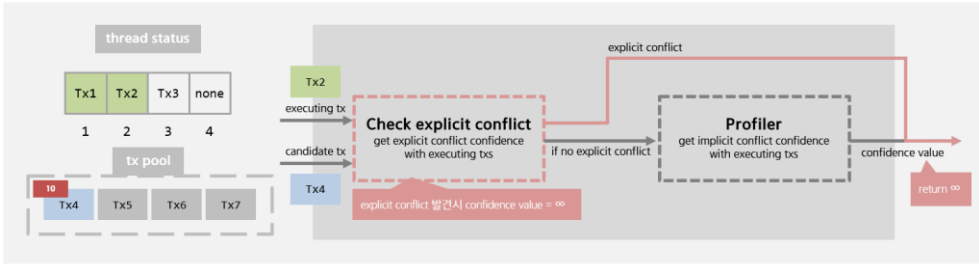
INC_RATE	DEC_RATE	THRESHOLD	SATURATED_WEIGHT
1	0.5	1	10

DEFAULT_OFFSET	CONFLICT_OFFSET	NONCONFLICT_OFFSET
0.1	1	-0.1

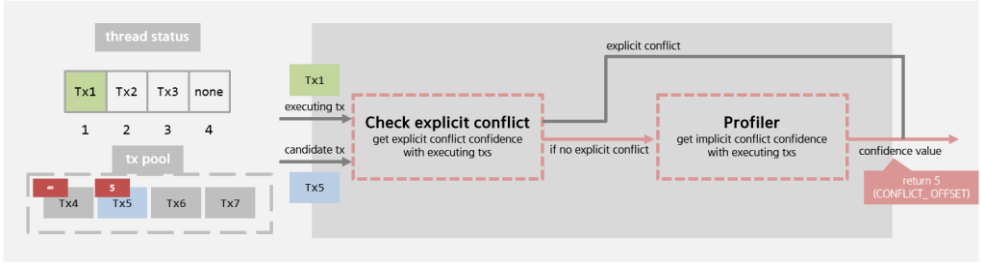
표 1 프로파일러 파라미터 설정값



(a) 프로파일 기반 스케줄러 구조



(b) Tx4의 컨피던스 값 계산 과정 중 명시적 충돌이 발견된 경우



(c) Tx5의 컨피던스 값 계산 과정 중 내재적 충돌이 예측된 경우

그림 9 프로파일 기반 스케줄러

제 5 장 실험 및 결과

제 1 절 시뮬레이션 설계

현재 이더리움 가상 머신은 멀티쓰레딩을 지원하고 있지 않기에, 제안하는 사전 스케줄링을 3장에서 언급한 실제 이더리움 워크로드에 대해 실험하기 위하여 한 블록에 대해 가상으로 트랜잭션들을 스케줄링하여 수행하는 시뮬레이션을 설계하였다. 4장에서 제안한 프로파일 기반 스케줄러를 통해 각 단위 시간마다 동기적으로 모든 쓰레드에 트랜잭션을 스케줄링 했고, 이를 수행했을 때의 커밋(commit), 중단(abort) 여부는 트랜잭션의 명시적 정보와 내재적 정보를 활용하여 2장에서 정의한 충돌의 정의에 따라 확인하였다. 또한 트랜잭션의 소요 시간은 각 트랜잭션의 실제 소모 가스에 비례한다고 가정하였다. 시뮬레이션에 사용된 프로파일러의 설정은 표 1과 같다.

제 2 절 결과

스케줄링 방식, 쓰레드 수, 프로파일러 충돌 그래프의 초기화 주기에 따른 속도 향상 정도를 각각 평가하였다. 속도 향상의 기준(baseline)은 싱글쓰레드로 순차 수행 했을 때 요구되는 소요시간이다.

스케줄링 방식에 따른 속도 향상

먼저 쓰레드 4개 기준으로 스케줄링 방식에 따른 속도 향상 정도를 살펴보면[그림 10(a)], 프로파일 기반 사전 스케줄링으로 동시 수행 했을 때 순차 수행 방식 대비 최대 3.1배, 기존의 그리디 스케줄링 방식 대비 최대 28.2%의 속도 향상을 보였다. 이러한 속도 향상은 최신 블록에 도달하며 점점 증가하는 양상을 보였는데, 그 이유는 블록체인 플랫폼이 크게 대중화되고 서비스되지 않았던 초반에는 한 블록 내에 동일한 수신자, 발신자를 가진 트랜잭션이 상당수를 차지하였기에 애초에 동시성이 높지 않았기 때문이다. 점점 다양한 트랜잭션이 발생하면서 오히려 동시 수행시의 충돌을 최소화하는 스케줄링의 여지가 생겨 사전 스케줄링의 효과가 증가한 것으로 예상된다. 비슷한 이유로, 명시적 충돌이 대부분을 차지하던 4번째 구간(3,000,000~4,000,000)까지는 프로파일러의 효과가 눈에 띄진 않지만, 복잡한 스마트 컨

트랙트가 도입되어가며 점점 내재적 충돌이 증가하는 최신 구간에 와서는 프로파일러가 없는 사전 스케줄링 대비 6% 정도의 속도 향상을 보였다. 실제로, 이러한 프로파일러의 효과 증가 추세는 3절에서 다룬 내재적 충돌 증가 추세와 비슷한 양상 지닌다. 게다가 앞으로도 이러한 내재적 충돌은 더욱 증가해갈 것으로 보이므로, 프로파일러의 영향 또한 더욱 중요해질 것이다.

쓰레드 수에 따른 속도 향상

그림 10(b)는 쓰레드 수에 따른 프로파일 기반 사전 스케줄링의 순차 수행 대비 속도 향상 효과를 나타낸다. 쓰레드 수를 32개까지 늘리더라도 이에 비례하는 속도 향상 효과는 나타나지 않았는데, 그 이유는 3장에서 살펴보았듯 애초에 한 블록 내의 트랜잭션 간 평균 충돌률이 64.5%로 높은 편이기에 동시성에 한계가 있기 때문이다. 또한 시뮬레이션에는 고려되지 않았지만, 쓰레드 수가 늘어날수록 많아지는 충돌로 인해 스케줄링 오버헤드 또한 늘어날 것이기에, 이더리움 워크로드의 충돌률을 고려한 적절한 쓰레드 수의 선정이 자원의 효율적인 활용에 중요할 것이다.

충돌 그래프 초기화 주기에 따른 속도 향상

그림 10(c)는 쓰레드 4개를 기준으로 한 프로파일러 충돌 그래프의 초기화 주기에 따른 사전 스케줄링 방식의 속도 향상 결과를 보여준다. 초기화 과정 없이 계속 데이터가 누적된 충돌 그래프의 경우, 시뮬레이션이 끝난 뒤 측정한 저장 용량[2]에서 약 250MB의 크기를 보였다. 본 시뮬레이션은 10,833,600 블록 데이터를 100 블록 단위로 샘플링하여 수행해본 것이기에, 이는 약 108,000여개 블록에 대한 충돌 그래프 용량으로 생각해볼 수 있다. 반면 10,000개 블록 단위로 그래프를 초기화한 경우 최대 그래프가 50MB 정도의 수준이었는데, 속도 향상 측면에서는 0.1% 안팎의 매우 근소한 차이이지만 오히려 좋은 효과를 보이는 구간이 있었다. 이렇게 주기적으로 충돌 그래프를 초기화했을 때 오히려 더 큰 속도 향상이 나타나는 이유는 그래프에서 과거 불필요한 정보를 주기적으로 삭제함으로써 오히려 최신 경향을 더욱 잘 반영했기 때문인 것으로 보인다. 또한 초기화 주기를 1,000 블록 또는 100

[2] 파이썬 pickle 라이브러리를 활용하여 저장 용량 측정

블록으로 설정하더라도 속도 향상 정도의 차이는 0.5% 정도에 불과하였는데, 이는 3절에서 분석했듯 소수의 주소 쌍이 대부분의 충돌을 일으키기에 적은 정보만 들고있어도 효율적으로 예측이 가능하기 때문이다. 심지어 초기화 주기가 1 블록인 경우는 아예 매 블록마다 충돌 그래프를 초기화한 것이라고 볼 수 있는데, 그럼에도 불구하고 프로파일러가 없는 사전 스케줄링과 비교하면 최대 2.6%의 속도 향상이 있었다. 초기화를 전혀 하지 않은 경우에도 최대 6% 정도의 속도 향상을 달성한 것을 감안하면 이는 스토리지 오버헤드 대비 높은 수치이다. 이렇듯 매 블록 충돌 그래프를 초기화 하더라도 프로파일링의 효과가 있는 이유는, 한 블록 내에서도 단위 시간마다 충돌 그래프가 업데이트되기에 한 블록 내에서의 중복된 충돌이 다시 발생하는 것을 방지했기 때문으로 보인다. 이러한 사실을 이용해 적절한 초기화 주기를 찾는다면, 충돌 그래프에 대한 스토리지 오버헤드와 탐색 수행 시간을 절약하면서, 속도 향상 측면에서도 큰 손해를 보지 않을 것이다.

후보 트랜잭션 수에 따른 속도 향상

이 전까지의 실험 결과는 트랜잭션 풀에 있는 모든 트랜잭션을 스케줄링 시의 후보 트랜잭션으로 간주한 결과이다. 이는 곧 트랜잭션 풀에 있는 모든 트랜잭션에 대해 컨피던스 값을 계산해야함을 뜻하며, 최악의 경우 $O(n^2)$의 비용이 들어가게 되기에[3] 오버헤드가 매우 클 것으로 예상된다. 따라서 후보 트랜잭션의 수를 2, 4, 16, 32 개로 고정하여 이에 따른 속도 향상 정도를 측정해보았다. 그 결과, 후보 트랜잭션을 32개로만 설정하여도 무한대의 후보 트랜잭션과 비교했을 때 속도 향상에는 불과 약 2%의 감소만이 있는 것을 확인할 수 있었다[그림 10(d)]. 이를 통해, 적절한 후보 트랜잭션 수를 설정하면 스케줄링으로 인한 시간적 오버헤드를 줄이면서도 속도 향상 측면에서 큰 손해를 보지 않을 것임을 알 수 있었다.

스케줄링 방법에 따른 스케줄링 정확도

마지막으로, 스케줄링 방법에 따른 스케줄링 정확도를 측정하였다[표 2]. 100 블록단위로 샘플링하여 9,000,000~9,999,800 블록에 대해 분석하였으며,

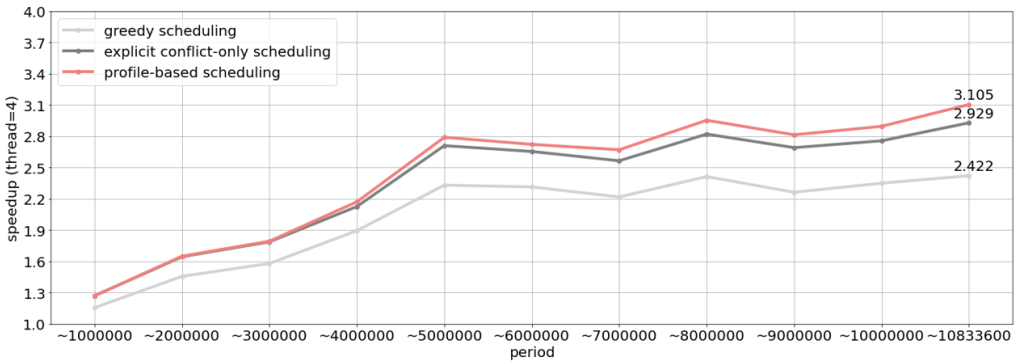
[3] 이 때, n은 해당 블록 안에 있는 트랜잭션 수

분석에 포함된 총 트랜잭션 수는 1,073,125 개이다. 스케줄링 정확도 계산을 위해 먼저 스케줄링에 대한 참, 거짓 케이스를 세 가지로 나눠 분류하였다. 스케줄링이 참인 경우로는 충돌이 발생하지 않는 올바른 트랜잭션을 스케줄링 하는 경우(케이스 1)와 충돌이 발생하지 않는 후보 트랜잭션이 없어 어떤 트랜잭션도 스케줄링 하지 않는 경우(케이스 2)가 있으며, 스케줄링이 거짓인 경우로는 충돌이 발생하는 트랜잭션을 스케줄링하는 경우(케이스 3)와 충돌이 발생하지 않는 후보 트랜잭션이 있으나 어떤 트랜잭션도 스케줄링 하지 않는 경우(케이스 4)가 있다. 하지만 본 연구에서 제안하는 투기적 사전 스케줄링의 경우, 모든 후보 트랜잭션에서 외재적 충돌이 예측되는 것이 아닌 이상, 충돌이 가장 적은 확률로 발생할 것으로 예측되는 트랜잭션, 즉 컨피던스 값이 가장 작은 트랜잭션을 무조건 스케줄링하므로 케이스 4의 경우는 발생하지 않으므로 결과에서 제외하였다. 따라서 프로파일러가 트랜잭션을 스케줄링 할 때마다 각 스케줄을 케이스 1, 2, 3 중 하나로 분류하여 각 케이스가 차지하는 비율을 계산함으로써 정확도를 계산해보았다.

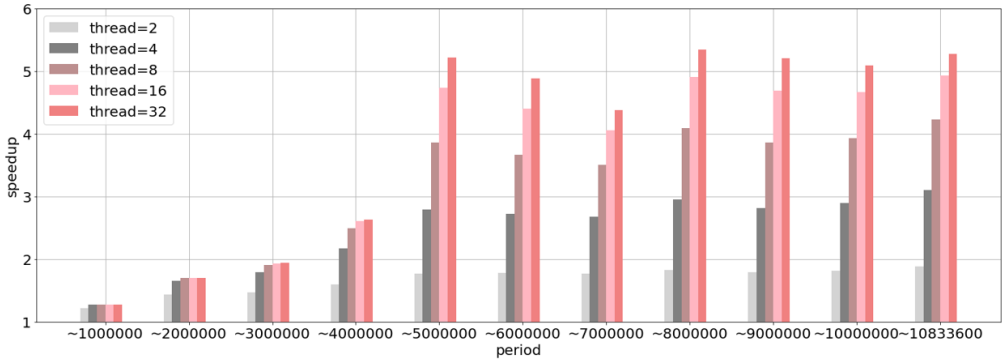
그 결과, 표 2에서 확인할 수 있듯, 프로파일러 기반 스케줄링의 false rate는 7%로, 그리디 스케줄링(49%) 또는 외재적 충돌만 고려한 스케줄링(12%) 대비 낮은 false rate를 보이는 것을 알 수 있었다. 이를 더욱 개선하기 위해서는 외재적 충돌만으로 내재적 충돌을 예측하는 과정을 더 정확히 해야할 것이며, 이를 위한 방법으로는 수신자 외의 다른 정보를 추가 기록한다거나 프로파일러의 파라미터를 조정해보는 방법이 있을 수 있다.

	True (case 1)	True (case 2)	False (case 3)	Total
Greedy	1,073,125(51%)	0(0%)	1,028,927(49%)	2,102,052
Explicit conflict only	1,073,125(68%)	310,614(20%)	193,151(12%)	1,576,890
Profile-based	1,073,125(74%)	282,641(19%)	96,715(7%)	1,452,481

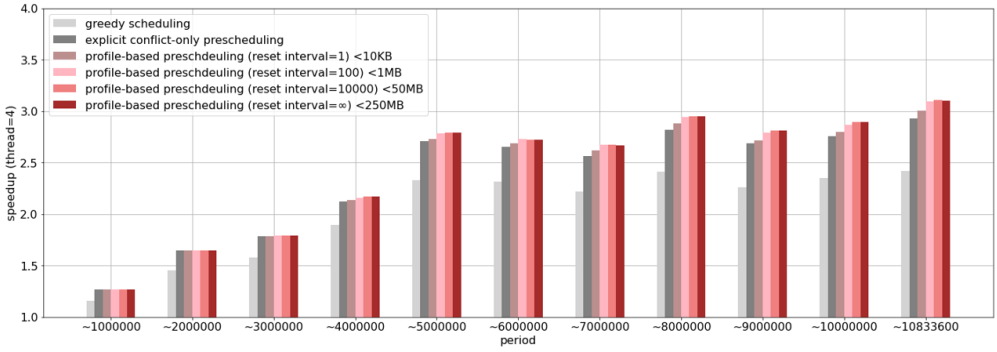
표 2 스케줄링 방법에 따른 스케줄링 정확도 분석



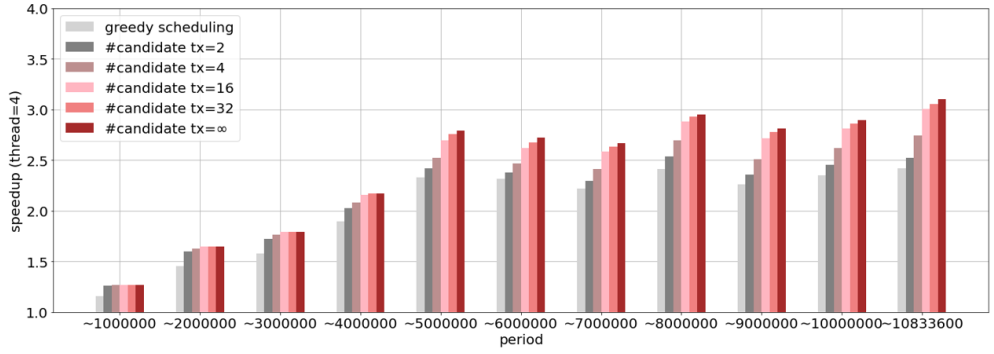
(a) 스케줄링 방식에 따른 블록 구간별 평균 속도 향상 (thread=4)



(b) 쓰레드 수에 따른 블록 구간별 평균 속도 향상



(c) 프로파일러 충돌 그래프 초기화 주기에 따른 블록 구간별 평균 속도 향상 (thread=4)



(d) 후보 트랜잭션 수에 따른 블록 구간별 평균 속도 향상 (thread=4)

그림 10 순차 수행 대비 속도 향상 결과

제 6 장 관련 연구

이더리움 워크로드 분석 연구

이 전까지의 이더리움 데이터 분석 연구는 동시 수행 상황을 가정하지 않았기에, 시간에 따른 트랜잭션 발생량의 변화나 자주 호출되는 스마트 컨트랙트에 대한 카테고리 분류 등의 분석[9][5]이 진행되었다. 특히 [9]에서는 이더리움의 계정을 노드로 하고, 트랜잭션 발생 관계나 이더를 주고받은 관계를 엣지로 나타내어 그래프를 분석하는 것에 초점을 두었다. [5]의 경우 스마트 컨트랙트를 유형별로 분류하고 이에 대한 통계를 중점으로 논의하였다.

이더리움 트랜잭션 동시 수행 연구

이더리움 트랜잭션 동시 수행에 관한 기존 연구들의 경우, 대부분 스마트 컨트랙트의 동시 수행에 초점을 맞추었다[6][10]. [6]에서는 전통적인 데이터베이스 분야에서 일관성을 유지하기 위해 사용하는 락 기반의 동시 수행 기법을 스마트 컨트랙트 동시수행 시 스토리지의 일관성 유지에 차용하였으며, [10]에서는 이러한 아이디어를 실제 이더리움 데이터에 적용할 경우 어느 정도의 속도 향상이 있을 지 시뮬레이션을 진행하였다. [11]의 경우 트랜잭션의 발신자와 수신자가 동일한 경우에 대해서 모두 충돌로 간주하여 속도 향상 정도를 수학적으로 모델링하는 연구를 진행하였다. 이러한 선행 연구들은 스토리지 동시 접근만을 충돌로 간주하거나 수·발신자가 같은 경우만 충돌로 간주함으로써 발생할 수 있는 모든 충돌을 고려하지 않았다는 한계가 있다.

사전 스케줄링 관련 연구

사전 스케줄링의 경우, 데이터베이스 분야의 트랜잭셔널 메모리의 동시 수행에 관한 연구인 [8]에서 제안된 바 있다. 벤치마크 수행 시, 대부분의 충돌은 결국 소수의 지점에서 발생하므로 이를 충돌 그래프와 쓰레드 상태 리스트를 두어 기록하여 예측에 활용하는 방식이다. 본 연구에서 이더리움 데이터에서 발생하는 대부분의 충돌은 결국 소수의 계정들에 의해 발생한다는 점과 유사하기에, 이러한 구조와 알고리즘을 프로파일러 구현에 활용하였다.

제 7 장 결론 및 향후 연구

점점 복잡해지고 많아져가는 트랜잭션에 비해 현재의 이더리움 순차 수행 디자인은 이더리움의 확장성을 제한하고있다. 본 논문에서는 이러한 한계점을 극복하고자 사전 스케줄링을 통한 트랜잭션 동시 수행을 제안했다. 그 결과, 트랜잭션에 명시된 발신자·수신자 정보를 통해 쉽게 예측할 수 있는 명시적 충돌뿐 아니라, 트랜잭션을 수행해보기 전까지는 알 수 없는 내재적 충돌까지도 과거 충돌 정보를 기록하는 프로파일러를 통해 예측함으로써, 쓰레드 4개를 기준으로 기존의 그리디한 방식의 동시 수행에 대비 수행 속도를 최대 28.2% 향상시켰으며, 순차 수행 대비 약 3.1배 향상시킬 수 있었다.

그러나 이러한 사전 스케줄링을 통한 동시 수행은 채굴 단계에서의 트랜잭션 동시 수행을 의미한다. 검증 단계에서의 트랜잭션 동시 수행을 다루기 위해서는 채굴 단계에서 블록에 담을 때 동시 수행된 트랜잭션들의 직렬 순서가 담겨야 할 것이며, 검증자들은 이러한 직렬 스케줄과 동등한 병렬 스케줄을 만들어내는 과정이 필요할 것이다. 채굴 단계뿐 아니라 검증 단계에서도 트랜잭션 동시 수행이 가능해진다면, 이더리움 네트워크에 새로운 노드가 참여할 때 필요한 검증 시간 또한 단축될 것이므로 이더리움이 지향하는 탈중앙성을 높이는 것에도 도움이 될 것이다. 따라서 이러한 부분이 향후 동시 수행의 연구 방향이 되어야 할 것이다.

참고 문헌

[1] Ethereum. Retrieved from https://github.com/ethereum/, 2020.

[2] Uniswap. Retrieved from https://uniswap.org/docs, 2020.

[3] Etherscan. The Ethereum Blockchain Explorer. Retrieved from https://etherscan.io/, 2020.

[4] G. A. Pierro and H. Rocha, "The Influence Factors on Ethereum Transaction Fees", 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019.

[5] M. Di Angelo and G. Salzer, "Characterizing Types of Smart Contracts in the Ethereum Landscape", 4th Workshop on Trusted Smart Contracts Financial Cryptography, 2020.

[6] T. Dickerson, P. Gazzillo, M. Herlihy and E. Koskinen, "Adding concurrency to smart contracts", Proc. ACM Symp. Principles Distrib. Comput., pp. 303-312, 2017.

[7] Ethereum Homestead Documentation. Retrieved from http://www.ethdocs.org/en/latest/, 2020.

[8] G. Blake, R. G. Dreslinski and T. Mudge, "Proactive transaction scheduling for contention management", Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 156-167, 2009.

[9] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, et al., "Understanding ethereum via graph analysis", IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pp. 1484-1492, 2018.

[10] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in Ethereum smart contracts", Proc. Int. Conf. Blockchain Econ. Secur. Protocols (Tokenomics), vol. 71, pp. 4:1-4:15, 2019.

[11] D. Reijsbergen and A. Dinh, "On Exploiting Transaction Concurrency To Speed Up Blockchains", Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS), 2020.

Abstract

Speculative Transaction Scheduling for Adding Concurrency to Ethereum

KOO Yeonjae

Dept. of Electrical and Computer Engineering

The Graduate School

Seoul National University

Ethereum is one of the most popular blockchain platform, and the amount of usage has been increasing for years based on interest in blockchain-based financial services. Despite the growing number of these complex transactions, Ethereum currently handles all transactions in a sequential execution method using a single thread, which limits throughput. To overcome this limitation, studies have been proposed to execute Ethereum transactions concurrently. However, if the transactions scheduled in a greedy manner, frequent collisions may lead to poor concurrency and high rollback costs.

In this paper, we propose a speculative scheduling technique that focuses on minimizing collisions between Ethereum transactions. Not only collisions that can be easily predicted by explicit feature in the transaction, but also collisions that can only detected by implicit feature are considered based on past records through profiling. As a result, the speedup is increased by 28.2% compared to the concurrent execution of the greedy method in 4-thread simulation, and the overall transaction execution speedup was improved by 3.1 times compared to sequential execution.

Keywords : Ethereum, Blockchain, Smart Contract, Transaction, Scheduling

Student Number : 2019-24567