Analysing Snapshot Isolation

Andrea Cerone
joint work with Alexey Gotsman

IMDEA Software Institute - Madrid, Spain

PaPoC - 2016, April 18th - Imperial College London
Snapshot Isolation

• Performs better than serialisability…

• …while still prohibiting several anomalies

• Provided by most commercial DBs
  *Oracle, Microsoft SQL server, postgreSQL, etc…*
This talk

• Original Specification of Snapshot Isolation

• Alternative Specification
  using Adya’s dependency graphs
  makes it easier to reason about program behaviour

• Transaction Chopping for Snapshot Isolation
Snapshot Isolation

- Transactions read data from a snapshot of the DB, taken at the moment they start

- Updates become visible to other transactions after commit
Snapshot Isolation

• Transactions read data from a snapshot of the DB, taken at the moment they start.

• Updates become visible to other transactions after commit.
Snapshots Isolation

- Transactions read data from a snapshot of the DB, taken at the moment they start.
- Updates become visible to other transactions after commit.

![Diagram showing the process of snapshot isolation with variables and operations.](chart)
Snapshots Isolation

- Transactions read data from a snapshot of the DB, taken at the moment they start.
- Updates become visible to other transactions after commit.

Write Conflict Detection

- Concurrent transactions write to one same object: at most one commits.
Write Skew Anomaly

Transaction mutual_withdraw1(int n) {
    if (acct1 + acct2 >= n)
        acct1 = acct1 - n;
}

Transaction mutual_withdraw2(int n) {
    if (acct1 + acct2 >= n)
        acct2 = acct2 - n;
}

start read(acct1,50) read(acct2,50) write(acct1,-10) commit

start read(acct1,50) read(acct2,50) write(acct2,-10) commit

acct1 = acct2 = 50

acct1 = acct2 = -10
Alternative Specification
Transactions

**Committed Transaction**

read($x$, 0): value fetched from the snapshot
write($y$, 1): final value written for the object
Run-time Dependencies (Adya, 1999)

S

\text{write}(x, 1) \xrightarrow{WR} \text{read}(x, 1)

T

T reads the value of $x$ from $S$

S

\text{write}(x, 1) \xrightarrow{WW} \text{write}(x, 2)

T

T overwrites the value of $x$ written by $S$
Run-time Anti-Dependencies

S reads a value for $x$ which is later updated by T
Theorem (Fekete et al. 2005):
\( A \) is an execution is in \( \text{SI} \)
All cycles in \( \text{DependencyGraph}(A) \)
have two adjacent \( \text{RW} \) edges

Application: Static Analysis for Robustness
A well Known Result

\[ \text{read(} \text{acct1}, 50) \quad \text{read(} \text{acct2}, 50) \quad \text{write(} \text{acct1}, -10) \]

\[\text{RW} \quad \text{RW}\]

\[ \text{read(} \text{acct1}, 50) \quad \text{read(} \text{acct2}, 50) \quad \text{write(} \text{acct2}, -10) \]
Our Contribution

Theorem (Fekete et al. 2005):

\( A \) is an execution is in \( \text{SI} \) \( \Rightarrow \)

All cycles in \( \text{DependencyGraph}(A) \) have two adjacent \( \text{RW} \) edges
Theorem (Fekete et al. 2005):
\( \mathcal{A} \) is an execution is in \( \text{SI} \) if and only if all cycles in \( \text{DependencyGraph}(\mathcal{A}) \) have two adjacent \( \text{RW} \) edges.

Application: Transaction Chopping for SI
Our Contribution

write($x, 1$) \xrightarrow{WR} \text{read}($x, 1$) \quad \text{read}($y, 0$)

\text{read}($x, 1$) \xrightarrow{WR} \text{read}($x, 0$) \quad \text{read}($y, 1$)

\text{write}($y, 1$) \xleftarrow{WR} \text{read}($x, 0$) \quad \text{read}($y, 1$)

\text{write}($x, 1$) \xleftarrow{WR} \text{read}($x, 1$) \quad \text{read}($y, 0$)
Transaction Chopping
Transaction Chopping

- Long Transactions are more likely to cause conflicts

```java
Transaction transfer(int acct1, int acct2, int n) {
    if (acct1 >= n) {
        acct1 = acct1 - n;  acct2 = acct2 + n;
    }
}```
Transaction Chopping

• Long Transactions are more likely to cause conflicts

• **IDEA**: chop transactions into chains of smaller ones

• *Chopping transactions can introduce new observable behaviour*

```c
Transaction withdraw(int acct1, int n) {
    if (acct1 >= n)
        acct1 = acct1 - n;
}
```

```c
Transaction deposit(int acct2, int n) {
    acct2 = acct2 + n;
}
```
Transaction Chopping

```java
Chain transfer(int n) {
    Transaction withdraw(n) {
        if (acct1 >= n)
            acct1 = acct1 - n;
    }
    Transaction deposit(n) {
        acct2 = acct2 + n;
    }
}
```

```
acct1 = 100   acct2 = 0

lookup : 100
transfer(50);
lookup : 100
acct1 = 50   acct2 = 50

Transaction lookup {
    return acct1 + acct2;
}
```
Transaction Chopping

Chain transfer(int n) {
    ...
}

Transaction withdraw(n) {
    if (acct1 >= n)
        acct1 = acct1 - n;
}

Transaction deposit(n) {
    acct2 = acct2 + n;
}

Transaction lookup {
    return acct1 + acct2;
}

acct1 = 100  acct2 = 0
withdraw(50);
lookup: 50
deposit(50);
acct1 = 50  acct2 = 50

acct1 = 100  acct2 = 0
withdraw(50);
acct1 = 50
acct2 = 50
Transaction lookup {
    return acct1 + acct2;
}
Chopping Graphs

Chain transfer(int n) {
    Transaction withdraw(n) {
        if (acct1 >= n)
            acct1 = acct1 - n;
    }
    Transaction deposit(n) {
        acct2 = acct2 + n;
    }
}

Transaction lookup {
    return acct1 + acct2;
}
Chopping Graphs

Chaining transfer(int n) {
  Transaction withdraw(n) {
    if (acct1 >= n)
      acct1 = acct1 - n;
  }
  Transaction deposit(n) {
    acct2 = acct2 + n;
  }
}

Transaction lookup {
  return acct1 + acct2;
}
Chopping Graphs

Chain transfer(int n) {
    Transaction withdraw(n) {
        if (acct1 >= n)
            acct1 = acct1 - n;
    }
    Transaction deposit(n) {
        acct2 = acct2 + n;
    }
}

Transaction lookup {
    return acct1 + acct2;
}
Chopping Graphs

Chain transfer(int n) {
  Transaction withdraw(n) {
    if (acct1 >= n)
      acct1 = acct1 - n;
  }

  Transaction deposit(n) {
    acct2 = acct2 + n;
  }
}

Transaction lookup {
  return acct1 + acct2;
}
Theorem: a transactional application can be chopped correctly under SI if its chopping graph has no simple cycle with at least one P edge, one WR/WW/RW edge and where RW edges are always separated by WR edges or WW edges.
Transaction deposit(n) {
    acct2 = acct2 + n;
}

Transaction withdraw(n) {
    if (acct1 >= n)
        acct1 = acct1 - n;
}

Transaction lookup1 {
    return acct1;
}

Transaction lookup2 {
    return acct2;
}

A Positive Example
A Positive Example

Chain transfer(int n) {

Transaction withdraw(n) {
    if (acct1 >= n)
        acct1 = acct1 - n;
}

Transaction deposit(n) {
    acct2 = acct2 + n;
}

Transaction lookup1 {
    return acct1;
}

Transaction lookup2 {
    return acct2;
}

Proof Strategy

```
Chain transfer(int n) {
    Transaction withdraw(n) {
        if (acct1 >= n)
            acct1 = acct1 - n;
    }

    Transaction deposit(n) {
        acct2 = acct2 + n;
    }

    Transaction lookup1 {
        return acct1;
    }

    Transaction lookup2 {
        return acct2;
    }
}
```
Proof Strategy

Fekete’s Criterion: Only cycles with adjacent RW edges
Proof Strategy

Chain transfer(int n) {
    Transaction withdraw(n) {
        if (acct1 >= n)
            acct1 = acct1 - n;
    }
    Transaction deposit(n) {
        acct2 = acct2 + n;
    }
}

Transaction lookup1 {
    return acct1;
}

Transaction lookup2 {
    return acct2;
}

read(acct1, 100) write(acct1, 50)
read(acct2, 0) write(acct2, 50)
read(acct1, 50)
read(acct1, 100) write(acct1, 50)
read(acct2, 0) write(acct2, 50)
read(acct1, 50)
Proof Strategy

Our Contribution

read(acct1, 100)  write(acct1, 50)
read(acct2, 0)    write(acct2, 50)
read(acct1, 50)   write(acct1, 50)
read(acct2, 50)   write(acct2, 50)
What to take away

• Dependency Graph Characterisation of SI

• Useful for reasoning about applications
  *Transaction Chopping, Robustness, etc.*

• Can be generalised to weaker consistency models