DoubleChecker: Efficient Sound and Precise Atomicity Checking

Swarndenu Biswas,
Jipeng Huang, Aritra Sengupta, and Michael D. Bond
The Ohio State University
PLDI 2014
Impact of Concurrency Bugs
Impact of Concurrency Bugs

Northeastern blackout, 2003
Nasdaq's Facebook Glitch Came From Race Conditions

Joab Jackson
@Joab_Jackson

May 21, 2012 12:30 PM

The Nasdaq computer system that delayed trade notices of the Facebook IPO on Friday was plagued by race conditions, the stock exchange announced Monday. As a result of this technical glitch in its Nasdaq OMX system, the market expects to pay out US$13 million or even more to traders.

A number of trading firms lost money due to mismatched Facebook share prices. About 30 million shares' worth of trading were affected, the exchange estimated.

On Friday, Nasdaq had delayed Facebook's IPO by 30 minutes. For about 20 minutes, the exchange stopped confirming trades placed by brokers, who were unable to see the results of their orders for more than two hours.
Atomicity Violations

- Constitute 69%\(^1\) of all non-deadlock concurrency bugs

---

Atomicity

- Concurrency correctness property
- Synonymous with **serializability**
  - Program execution must be equivalent to some serial execution of the atomic regions
Atomicity Violation Example

Thread 1

```java
void execute() {
    while (...) {
        prepareList();
        . . . .
        processList();
        . . . .
        resetList();
    }
}
```

Thread 2

```java
void execute() {
    while (...) {
        prepareList();
        . . . .
        processList();
        . . . .
        resetList();
    }
}
```
Atomicity Violation Example

Thread 1

```java
void prepareList() {
    synchronized (l1) {
        list.add(new Object());
    }
}

void processList() {
    synchronized (l1) {
        Object head = list.get(0);
    }
}
```

Thread 2

```java
void resetList() {
    synchronized (l1) {
        list = null;
    }
}
```
Atomicity Violation Example

Thread 1

```java
void prepareList() {
    synchronized (l1) {
        list.add(new Object());
    }
}
```

```java
void processList() {
    synchronized (l1) {
        Object head = list.get(0);
    }
}
```

Thread 2

```java
void resetList() {
    synchronized (l1) {
        list = null;
    }
}
```

Null pointer dereference

Data-race-free program

Atomicity Violation Example
Atomicity Violation Example

Thread 1
void execute() {
    while (...) {
        prepareList();
        processList();
        resetList();
    }
}

Thread 2
void execute() {
    while (...) {
        prepareList();
        processList();
        resetList();
    }
}
Detecting Atomicity Violations

- Check for conflict serializability
  - Build a transactional dependence graph
  - Check for cycles
- Existing work
  - Velodrome, Flanagan et al., PLDI 2008
  - Farzan and Parthasarathy, CAV 2008
Transactional Dependence Graph

Thread 1:
- Transaction
- Time
- acq lock
- wr o.f
- rel lock

Thread 2:
- wr o.g

Thread 3:
- wr o.f
Transactional Dependence Graph

- Thread 1:
  - acq lock
  - wr o.f
  - rel lock

- Thread 2:
  - wr o.g

- Thread 3:
  - wr o.f
Cycle means Atomicity Violation
Prior Work is Slow

Velodrome\textsuperscript{1}

- Paper reports 12.7X overhead
- 6.1X in our experiments

\textsuperscript{1} C. Flanagan et al. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In PLDI, 2008.
High Overheads of Prior Work

- Precise tracking is expensive
  - “last transaction(s) to read/write” for every field
  - Need atomic updates in instrumentation
Instrumentation Approach

Uninstrumented program

Instrumented program
Precise Tracking is Expensive!

Program access

Precise tracking of dependences

Analysis-specific work

Update metadata

Program access

Can lead to remote cache misses for mostly read-only variables

Uninstrumented program

Instrumented program
Synchronized Updates are Expensive!

Uninstrumented program

Program access

atomic

Lock metadata access

Unlock metadata access

Program access

Instrumented program
Synchronized Updates are Expensive!

- Synchronization on every access slows programs.

- Lock metadata access
  - atomic

- Program access
  - atomic

- Unlock metadata access
  - Instrumented program

- Program access
  - Uninstrumented program
DoubleChecker
DoubleChecker’s Contributions

- Dynamic atomicity checker based on conflict serializability
- Precise
  - Sound and unsound operation modes
- Incurs 2-4 times lower overheads
- Makes dynamic atomicity checking more practical
Key Insights

- Avoid **high costs** of precise tracking of dependences at every access
  - Common case: no dependences
    - Most accesses are thread local
Key Insights

- Tracks dependences **imprecisely**
  - Soundly over-approximates dependences
  - Recovers precision when required
  - Turns out to be a lot **cheaper**
Staged Analysis

- Imprecise cycle detection (ICD)
- Precise cycle detection (PCD)
Imprecise Cycle Detection

- Processes every program access
- Soundly overapproximates dependences, is cheap
- Could have false positives
Precise Cycle Detection

- Processes a subset of program accesses
- Performs precise analysis
- No false positives
Staged Analyses: ICD and PCD
Program execution to ICD has atomicity specifications, leading to sound tracking in Imprecise cycles. Precise violations go to PCD for static program locations and access information. ICD is Sound.
Role of ICD

- Most accesses in a program are thread-local
  - Uses Octet\textsuperscript{1} for tracking cross-thread dependences
- Acts as a dynamically sound transaction filter

---
\textsuperscript{1} M. Bond et al. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In OOPSLA, 2013.
Role of PCD

- Processes transactions involved in an ICD cycle
  - Performs precise serializability analysis
  - PCD has to do much less work
    - Program conforming to its atomicity specification will have very few cycles
Different Modes of Operation

- Single-run mode
- Multi-run mode
Single-Run Mode

Program execution → ICD+PCD → Atomicity violations

ICD cycles → PCD

Atomicity specifications

read/write logs
Multi-run Mode

Program execution

ICD

Program execution

ICD+PCD

First run

Potentially imprecise cycles

Static transaction information

Second run

Atomicity violations
Design Choices

● Multi-run mode
  ○ Conditionally instruments non-transactional accesses
    ■ Otherwise overhead increases by 29%
  ○ Could use Velodrome for the second run
    ■ But performance is worse
      • Second run has to process many accesses
      • ICD is still effective as a dynamic transaction filter
Examples

- Imprecise analysis
- Precise analysis
Imprecise Analysis

Thread 1

Thread 2

Thread 3

Thread 4

transaction

time

wr o.f
(\text{WrEx}_{T1})
Imprecise Analysis

- wr o.f (WrEx_{T1})

Thread 1
Thread 2
Thread 3
Thread 4
Imprecise Analysis
Imprecise Analysis

Thread 1

wr o.f
(WrEx_{T1})

time

Thread 2

rd o.g
(RdEx_{T2})

rd o.f
(RdSh_c)

Thread 3

Thread 4
Imprecise Analysis

Thread 1

\textbf{wr o.f}
\begin{align*}
&\text{(WrEx}_{T_1}\text{)} \\
\end{align*}

Thread 2

\textbf{rd o.g}
\begin{align*}
&\text{(RdEx}_{T_2}\text{)} \\
\end{align*}

Thread 3

\textbf{rd o.f}
\begin{align*}
&\text{(RdSh}_{c}\text{)} \\
\end{align*}

Thread 4

\textbf{rd o.h}
\begin{align*}
&\text{(fence)} \\
\end{align*}

\textbf{time}
Imprecise Analysis
Thread 1: wr o.f
Thread 2: rd o.g
Thread 3: rd o.f
Thread 4: rd o.h

Precise Analysis
No Precise Violation
Precise Violation
Evaluation Methodology

- Implementation
- Atomicity specifications
- Experiments
Implementation

● DoubleChecker and Velodrome
  ○ Developed in Jikes RVM 3.1.3
  ○ Artifact successfully evaluated
  ○ Code shared on Jikes RVM Research Archive
Experimental Methodology

● Benchmarks
  ○ DaCapo 2006, 9.12-bach, Java Grande, other benchmarks used in prior work\(^1\)

● Platform: 3.30 GHz 4-core Intel i5 processor

---

Atomicity Specifications

- Assume provided by the programmers
- We reuse prior work’s approach to infer the specifications

All methods except main(), run(), callers of join(), wait(), etc. considered non-atomic

DoubleChecker/Velodrome

new violations reported?

Yes

No

atomicity specification
Soundness Experiments

- Generated atomicity violations with
  - Velodrome - sound and precise
  - DoubleChecker
    - Single-run mode - sound and precise
    - Multi-run mode - unsound
- Results match closely for Velodrome and the single-run mode
  - Multi-run mode finds 83% of all violations
Performance Experiments
Performance Experiments

- **Single-run mode** - 1.9 times faster than Velodrome
- **Multi-run mode**
  - First run - 5.6 times faster
  - Second run - 3.7 times faster
DoubleChecker

- 2-4 times lesser overhead than current state-of-art
- Makes dynamic atomicity checking more practical
Related Work

● **Type systems**
  - Flanagan and Qadeer, PLDI 2003
  - Flanagan et al., TOPLAS 2008

● **Model checking**
  - Farzan and Madhusudan, CAV 2006
  - Flanagan, SPIN 2004
  - Hatcliff et al., VMCAI 2004
Related Work

● Dynamic analysis
  ○ Conflict-serializability-based approaches
    ■ Flanagan et al., PLDI 2008; Farzan and Madhusudan, CAV 2008
  ○ Inferring atomicity
    ■ Lu et al., ASPLOS 2006; Xu et al., PLDI 2005; Hammer et al., ICSE 2008
  ○ Predictive approaches
    ■ Sinha et al., MEMOCODE 2011; Sorrentino et al., FSE 2010
  ○ Other approaches
    ■ Wang and Stoller, PPoPP 2006; Wang and Stoller, TSE 2006
What Has DoubleChecker Achieved?

- **Improved overheads** over current state-of-art
  - Makes dynamic atomicity checking more practical
- **Cheaper to over-approximate dependences**
  - Showcases a judicious separation of tasks to recover precision