GPredict: Generic Predictive Concurrency Analysis

Jeff Huang, Qingzhou Luo, Grigore Rosu

Texas A&M University, University of Illinois at Urbana-Champaign
Concurrency Bugs

Data races

Atomicity Violations

Deadlocks

Order Violations

Atomic-set Serializability Violations

...
Real Problems Caused by Concurrency Bugs

Therac-25  3+ dead

$500M losses for investors

50 million people in dark

North American Blackout of 2003
Predictive Trace Analysis

program → Dynamic Analysis
Collect shared data accesses & synchronization operations

trace → Static Analysis
Explore thread scheduling freedom

bugs

Chen et al. 2008
Wang et al. 2009
Huang et al. 2011
Huang et al. 2014
Predictive Trace Analysis

**Program** → **Dynamic Analysis** → **Trace** → **Static Analysis** → **Bugs**

- **Dynamic Analysis**
  - Collect shared data accesses & synchronization operations
- **Static Analysis**
  - Explore thread scheduling freedom

*More precise than static analysis*
*More powerful than dynamic analysis*

**References**
- Chen et al. 2008
- Wang et al. 2009
- Huang et al. 2011
- Huang et al. 2014
Predictive Trace Analysis

![Diagram](image)

**y is volatile**

**Thread T1**
1: \( x = 1 \)
2: \( y = 1 \)
3: if(\( x < 0 \))
4: **Error**

**Thread T2**
5: \( y = 0 \)
6: if(\( y == 1 \))
7: \( x = -1 \)
### Predictive Trace Analysis

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: x=1</td>
<td>5: y=0</td>
</tr>
<tr>
<td>2: y=1</td>
<td>6: if(y==1)</td>
</tr>
<tr>
<td>3: if(x&lt;0)</td>
<td>7: x=-1</td>
</tr>
</tbody>
</table>

#### Race: (3,7)

* y is volatile
Predictive Trace Analysis

y is volatile

Thread T1
1: x = 1
2: y = 1
3: if(x < 0)
4: Error

Thread T2
5: y = 0
6: if(y == 1)
7: x = -1

Race: (3,7)
Not race: (1,7)
Predictive Trace Analysis

schedule a: 1-5-2-3-6-7

Thread T1
1: x=1
2: y=1
3: if(x<0)
4: Error

Thread T2
5: y=0
6: if(y==1)
7: x=-1
Predictive Trace Analysis

**Schedule a:** 1-5-2-3-6-7

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: x=1</td>
<td>5: y=0</td>
</tr>
<tr>
<td>2: y=1</td>
<td>6: if(y==1)</td>
</tr>
<tr>
<td>3: if(x&lt;0)</td>
<td>7: x=-1</td>
</tr>
<tr>
<td>4: <strong>Error</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Race:** (3,7)

**Not race:** (1,7)
Predictive Trace Analysis

(schedule b: 1-5-2-6-7-3-4)

Thread T1
1: x=1
2: y=1
3: if(x<0)
4: Error

Thread T2
5: y=0
6: if(y==1)
7: x=-1

Race: (3,7)
Creating an execution to trigger the error

Huang et al. 2011, 2014
Concurrency Bugs

Data races
Order Violations
Atomicity
Atomic-set Violations
Deadlocks
...

All are low-level memory access errors!
High-level Property

Authenticate-before-use:

A method authenticate must be always called before a method use that uses a resource
High-level Property

Authenticate-before-use:

A method **authenticate** must be always called **before** a method **use** that uses a resource

No shared data!
High-level Property

Authenticate-before-use:

A method `authenticate` must be always called before a method `use` that uses a resource

No shared data!

Safe iterator:

A Java `collection` is not allowed to be modified when an `iterator` is accessing its elements
High-level Property

Authenticate-before-use:

A method authenticate must always be called

Existing low-level memory error-based detectors cannot detect these high-level property violations

A Java collection is not allowed to be modified when an iterator is accessing its elements
Contributions

A generic predictive trace analysis technique to detect both high-level property violations and low-level memory errors

- A specification language for defining concurrency property violations
- An efficient constraint encoding approach based on thread local traces
Key Idea

Uniformly model property violations and thread schedules as simple ordering constraints and detect violations through constraint solving

Give an ORDER variable $O$ to each critical event:

**Authenticate-before-use**

$O_{auth}$: order of the “authentication method call” event

$O_{use}$: order of a “resource use method call” event

$O_{use} < O_{auth}$
Key Idea

Uniformly model property violations and thread schedules as simple ordering constraints and solve them with an SMT solver.

Give an ORDER variable $O$ to each critical event:

Authenticate-before-use

$O_{\text{auth}}$: order of the “authentication method call” event

$O_{\text{use}}$: order of a “resource use method call” event

$O_{\text{use}} < O_{\text{auth}}$

Safe iterator

$O_{\text{create}} < O_{\text{update}} < O_{\text{next}}$
An Example

Collection<Item> c;
Item A, B; Iterator i₁, i₂;

T₁
1: c.add(A);
2: start T2;
3: i₁=c.iterator();
4: i₁.next()  

T₂
5: c.add(B);
6: i₂=c.iterator();
7: i₂.next()

Oi - order of event at line i
An Example

```
Collection<Item> c;
Item A, B; Iterator i1, i2;

T1
1: c.add(A);
2: start T2;
3: i1=c.iterator();
4: i1.next();

T2
5: c.add(B);
6: i2=c.iterator();
7: i2.next();
```

Property violation constraint: 

$O_3 < O_5 < O_4$
An Example

Collection<Item> c;
Item A, B; Iterator i1, i2;

T1
1: c.add(A);
2: start T2;
3: i1=c.iterator();
4: i1.next()

T2
5: c.add(B);
6: i2=c.iterator();
7: i2.next()

Oi - order of event at line i
O3 < O5 < O4
O1 < O2 < O3 < O4
O5 < O6 < O7
O2 < O5

Property violation constraint:
Thread scheduling constraints:
An Example

Collection<Item> c;
Item A, B; Iterator i₁, i₂;

T₁
1: c.add(A);
2: start T2;
3: i₁=c.iterator();
4: i₁.next();

T₂
5: c.add(B);
6: i₂=c.iterator();
7: i₂.next();

Oi - order of event at line i

O₁ < O₂ < O₃ < O₄
O₅ < O₆ < O₇
O₂ < O₅

Property violation constraint:
Thread scheduling constraints:

A property violation schedule:
1->2->3->5 ->4->6->7
GPredict Overview

program

properties

Tracer

Constraint Encoder

SMT Solver

JavaMOP
[Chen & Rosu 2007]

violations

[Chen & Rosu 2007]
GPredict Overview

Predictive Runtime Verification

[Chen & Rosu 2007]

program

properties

violation
Challenges

**Usability**: how to define the property violations and easily use the technique in practice?

**Soundness & Completeness**: we want to find as many as possible bugs with no false alarms

**Runtime Performance**: reduce logging overhead as much as possible

**Scalability**: how to efficiently solve the constraints?
Challenges

**Usability**: how to define the property violations and use the technique in practice?

**Soundness & Completeness**: we want to find as many as possible bugs with no false alarms

**Runtime Performance**: reduce logging overhead as much as possible

**Scalability**: how to efficiently solve the constraints?
Challenges

**Usability**: how to define the property violations and

RegExp patterns & JavaMOP extension

**Soundness & Completeness**: we want to find as

Parametric constraint encoding

**Runtime Performance**: reduce logging overhead as much as possible

**Scalability**: how to efficiently solve the constraints?
Challenges

**Usability**: how to define the property violations and

RegExp patterns & JavaMOP extension

**Soundness & Completeness**: we want to find as

Parametric constraint encoding

**Runtime Performance**: reduce logging overhead as

Constraint encoding with thread local traces

**Scalability**: how to efficiently solve the constraints?
Challenges

**Usability**: how to define the property violations and easily use the technique in practice?

- RegExp patterns & JavaMOP extension

**Soundness & Completeness**: we want to find as many as possible bugs with no false alarms?

- Parametric constraint encoding

**Runtime Performance**: reduce logging overhead as much as possible?

- Constraint encoding with thread local traces

**Scalability**: how to efficiently solve the constraints?

- Existing high performance solvers: Z3 & Yices
### Property Specification Syntax

```
<GPredict Specification> ::= <Property Name> "(" <Parameters> ")" "{" <Event>* <Pattern> "}" 
<Event> ::= "event" <Id> <AspectJ AdviceSpec> "::" <AspectJ Pointcut> 
<Pattern> ::= "pattern:" (<RegExp> "|" <RegExp>)* <RegExp> 
<Property Name> ::= <Identifier> 
<Parameters> ::= (<Type> <Identifier>)+ 
(Id) ::= <Identifier> 
<AspectJ AdviceSpec> ::= AspectJ AdviceSpec syntax 
<AspectJ Pointcut> ::= AspectJ Pointcut syntax 
<Thread> ::= <Identifier> 
<AtomRegion> ::= <Identifier> 
<Identifier> ::= Java Identifier syntax 
<Begin> ::= "<" <AtomRegion> 
<End> ::= ">" <AtomRegion> 
<RegExp> ::= Regular expression over \{<Id>, <Id>"("<Thread>,<Begin> | <End>"\)}
```

Gray color: new syntax introduced for concurrency properties
Support RegExp patterns, syntax based on JavaMOP & AspectJ
Property Specification Syntax

DataRace (Object o) {
    event read before(Object o):
        get(* s) && target(o);
    event write before(Object o):
        set(* s) && target(o);

    pattern: read(t1) || write(t2)
}

Gray color: new syntax introduced for concurrency properties
Support RegExp patterns, syntax based on JavaMOP & AspectJ
Gray color: new syntax introduced for concurrency properties

Support RegExp patterns, syntax based on JavaMOP & AspectJ

Property Specification Syntax

```java
UnsafeIterator (Collection c, Iterator i) {
    event create after(Collection c) returning(Iterator i) :
        call(Iterator Collection+.iterator()) && target(c);
    event update after(Collection c) :
        (call(* Collection+.remove*(..)) || call(* Collection+.add*(..)) ) && target(c);
    event next before(Iterator i) :
        call(* Iterator.next()) && target(i);

    pattern: create next* update+ next
}
```
In a database application, a table must be created before any update or query on the table, and all operations must finish before closing the database connection.

UnsafeDatabaseAccess(Connection conn, String table) {
    event open after() returning(Connection conn):
        call(Connection DriverManager.getConnection(String));
    event close before(Connection conn):
        call(void Connection.close())&&target(conn);
    event create before(Connection conn, String table):
        call(* createTable(Connection conn, String table))&&args(conn,table);
    event delete before(Connection conn, String table):
        call(* deleteTable(Connection conn, String table))&&args(conn,table);
    event update before(String table, String sql):
        call(boolean Statement.execute(String))
        &&args(sql)&&if(sql.contains(table));

    pattern : !(open create update delete close)
Parametric Encoding

Binding property parameters to concrete object instances

Caveat: an identifier may match with multiple events

UnsafeIterator pattern: create next* update+ next

<table>
<thead>
<tr>
<th>Item A, B;</th>
<th>Iterator i1, i2;</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: c.add(A);</td>
<td></td>
<td>5: c.add(B);</td>
<td></td>
</tr>
<tr>
<td>2: start T2;</td>
<td></td>
<td>6: i2=c.iterator();</td>
<td></td>
</tr>
<tr>
<td>3: i1=c.iterator();</td>
<td></td>
<td>7: i2.next();</td>
<td></td>
</tr>
<tr>
<td>4: i1.next();</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parametric Encoding

Binding property parameters to concrete object instances

Provided by JavaMOP

Caveat: an identifier may match with multiple events

UnsafeIterator pattern: create next* update+ next

Item A, B; Iterator i1, i2;

T1
1: c.add(A);
2: start T2;
3: i1=c.iterator();
4: i1.next()

T2
5: c.add(B);
6: i2=c.iterator();
7: i2.next()

O3 < O1 < O4
O1 < O2 < O3 < O4
O5 < O6 < O7
O2 < O5

Caveat: an identifier may match with multiple events
Parametric Encoding

Binding property parameters to concrete object instances

Provided by JavaMOP

Caveat: an identifier may match with multiple events

UnsafeIterator pattern: create next* update+ next

Item A, B;  Iterator i1, i2;

Solution: Enumerate all matching events and disjunct constraints

4: i1.next()

7: i2.next()
Thread Local Encoding

Predictive trace analysis often requires a global trace
But, logging a globally-ordered trace of events is expensive
Thread Local Encoding

Predictive trace analysis often requires a global trace
But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log
Thread Local Encoding

Predictive trace analysis often requires a global trace
But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log

Thread scheduling constraints

\[ \Phi = \Phi_{mhb} \land \Phi_{sync} \land \Phi_{rw} \]

Must happens-before constraints \( (\Phi_{mhb}) \)
Synchronization Constraints \( (\Phi_{sync}) \)
Read-write constraints \( (\Phi_{rw}) \)
Thread Local Encoding

Predictive trace analysis often requires a global trace
But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log

Thread scheduling constraints

\[ \Phi = \Phi_{mhb} \land \Phi_{sync} \land \Phi_{rw} \]

- Must happens-before
- Synchronization Constraints
- Read-write constraints

All take thread local traces as input
## Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
<th>#Threads</th>
<th>#Events</th>
<th>#Violations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>UnsafeIterator</td>
<td>21</td>
<td>230</td>
<td>20</td>
<td>0.85s</td>
</tr>
<tr>
<td>H2</td>
<td>UnsafeDBAccess</td>
<td>5</td>
<td>126</td>
<td>16</td>
<td>0.67s</td>
</tr>
<tr>
<td>Derby1</td>
<td>NullPointerDeref</td>
<td>3</td>
<td>12K</td>
<td>5</td>
<td>4.7s</td>
</tr>
<tr>
<td>Derby2</td>
<td>CheckThenAct</td>
<td>3</td>
<td>21K</td>
<td>4</td>
<td>8.5s</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>DataRace</td>
<td>12</td>
<td>17K</td>
<td>29</td>
<td>17.6s</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>AtomicityViolation</td>
<td>3</td>
<td>64</td>
<td>2</td>
<td>0.4s</td>
</tr>
<tr>
<td>JDK-Logger</td>
<td>Deadlock</td>
<td>2</td>
<td>570</td>
<td>1</td>
<td>2.6s</td>
</tr>
</tbody>
</table>

Predicted **77** violations in seven benchmarks with real bugs.

More description of the results: [http://parasol.tamu.edu/~jeff/gpredict/](http://parasol.tamu.edu/~jeff/gpredict/)
# Runtime Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>#Threads</th>
<th>Base Time</th>
<th>Global</th>
<th>Local (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>4</td>
<td>1.8s</td>
<td>2m20s</td>
<td>2m6s (10%)</td>
</tr>
<tr>
<td>Batik</td>
<td>2</td>
<td>2.1s</td>
<td>2m28s</td>
<td>1m2s (58%)</td>
</tr>
<tr>
<td>Xalan</td>
<td>11</td>
<td>1.7s</td>
<td>6m14s</td>
<td>1m9s (82%)</td>
</tr>
<tr>
<td>Lusearch</td>
<td>10</td>
<td>1.4s</td>
<td>15m17s</td>
<td>23m16s (78%)</td>
</tr>
<tr>
<td>Sunflow</td>
<td>25</td>
<td>1s</td>
<td>19m48s</td>
<td>11m28s (42%)</td>
</tr>
</tbody>
</table>

Runtime overhead reduced by **54%** on average
Summary

Generic predictive trace analysis for concurrent programs base on constraint solving

- Predict both high-level property violations and low-level memory access errors

Predictive runtime verification

- A concurrency property specification language
- Constraint encoding with thread local traces -- significant runtime performance improvement
GPredict: Generic Predictive Concurrency Analysis

Jeff Huang, Qingzhou Luo, Grigore Rosu
Texas A&M University  University of Illinois at Urbana-Champaign

Thank You & Questions?