RACE DETECTION FOR EVENT DRIVEN APPLICATIONS

Veselin Raychev  Martin Vechev  Manu Sridharan
ETH Zurich  ETH Zurich  IBM T. J. Watson
Event-Driven: Motivation

~ 1 trillion websites today

~ 1 billion smartphones by 2016

Reacts to events: user clicks, arrival of network requests
Event-Driven: Motivation

Wanted: fast response time

- 1 trillion websites today
- 1 billion smartphones by 2016

Reacts to events: user clicks, arrival of network requests
Event-Driven: Motivation

~ 1 trillion websites today

Wanted: fast response time

~ 1 billion smartphones by 2016

Highly Asynchronous, Complex control flow
Non-determinism: network latency
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">

</body>
</html>
```

Gates = great
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">  
<img src="img2.png" onload="alert(Gates);">  

</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';"> 
<img src="img2.png" onload="alert(Gates);"> 

Gates = great
fetch img1.png
fetch img2.png

img2.png loaded
great is read from Gates
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>

<body>
<script>
var Gates = "great";</script>

<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">

</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';"/>
<img src="img2.png" onload="alert(Gates);"/>
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';"><br/>
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';">  
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```
Non-determinism: network latency

```html
<html>
<head></head>
<body>
<script>
var Gates = "great";
</script>
<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">
</body>
</html>
```

Gates = great
fetch img1.png
img1.png loaded
Gates = poor
fetch img2.png
img2.png is loaded
poor is read from Gates
Non-determinism: user interaction

```
<html><body>
// Lots of code
<input type="button" id="b1" onclick="javascript:f();">
// Lots of code
<script>
    f = function() {
        alert("hello");
    }
</script>
...
</body></html>
```
Non-determinism: user interaction

```html
<html>
<body>

// Lots of code

<input type="button" id="b1" onclick="javascript:f()">

// Lots of code

<script>
  f = function() {
    alert("hello");
  }
</script>

...</body></html>

User

parse <input>
```
Non-determinism: user interaction

```html
<html><body>

// Lots of code

<input type="button" id="b1" onclick="javascript:f()">

// Lots of code

<script>
  f = function() {
    alert("hello");
  }
</script>

...</body></html>
```

User

parse <input>

click button
read("f"),
crash

Non-determinism: user interaction
Non-determinism: user interaction

```html
<html><body>
// Lots of code
<input type="button" id="b1" onclick="javascript:f()">

// Lots of code
<script>
    f = function() {
        alert("hello");
    }
</script>
...
</body></html>
```

User

- Click button
- Read("f")
- Crash
What do we learn from these?

- Asynchrony causes non-determinism which may cause unwanted behavior

- Non-determinism is caused by interfering unordered accesses to shared locations
  - can be seen as data races

Can we detect such data races?
Race Detection Template

- Concurrent program
- Happens-Before Model
- Memory Locations

Race Detector

race 1
race 2
race 3
....
race N
Race Detection: Web

[Diagram with unclear symbols and text]
Memory locations

- "Normal", C-like, memory locations for JavaScript variables
- Functions are treated like "normal" locations
- HTML DOM elements
- Event, event-target and event-handler tuple
What is an atomic action?
- E.g.: parsing a single HTML element, executing a script, processing an event handler

How to order actions?
- E.g.: parsing of HTML elements of the page is ordered

Laborious to define: go over HTML5 spec
- Browser differences...
Example of Happens - Before

```html
<html>
<head></head>
<body>

<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">

<img src="img2.png" onload="alert(Gates);">

</body>
</html>
```
Example of Happens - Before

```html
<html>
<head></head>
<body>

<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">
<img src="img2.png" onload="alert(Gates);">

</body>
</html>
```
Example of Happens - Before

```html
<html>
<head></head>
<body>

<script>
var Gates = "great";
</script>

<img src="img1.png" onload="Gates='poor';">

<img src="img2.png" onload="alert(Gates);">

</body>
</html>
```
Happens - Before Model

Memory Locations

We will explore dynamic race detectors
Race Detection for Web: Challenges

- **Precision**: state-of-the-art detectors lead to too many *false positives*
  - caused by synchronization with read/writes, very common on the Web

- **Scalability**: state-of-the-art race detectors do not scale
  - blow-up in size of data structures caused by too many event handlers
Race Detection for Web: Challenges

- **Precision:** state-of-the-art detectors lead to too many *false positives*
  - caused by synchronization with read/writes, very common on the Web

- **Scalability:** state-of-the-art race detectors do not scale
  - blow-up in size of data structures caused by too many event handlers
Precision Issue: Example

3 variables with races:
- `init`
- `y`
- `y.g`

Some races are synchronization:
- `init`

Reports false races:
- `y`
- `y.g`
Wanted: “guaranteed” races

Intuition: identify races that are guaranteed to exist.

We want to report races on variable `init`

But not on:

- `y`
- `y.g`

Because fixing the races on `init` will always remove all races on `y` and `g` (in this trace).
Wanted: “guaranteed” races

A race (c, d) is **guaranteed** if in one trace we see c ... d and in another trace we see d ... c

Here, race (c,d) is **guaranteed**
Wanted: “guaranteed” races

A race (c, d) is **guaranteed** if in one trace we see c … d and in another trace we see d … c

Approach: record the full program trace and then compute data-dependence, etc…

Expensive!
Wanted: “guaranteed” races

An abstraction of the program trace:

- **a**: read(y)
- **b**: read(y.g)
- **c**: write(init)
- **d**: read(init)

Common approach: record only shared reads and writes.
Wanted: “guaranteed” races

An abstraction of the program trace:

Common approach: record only shared reads and writes.
Wanted: “guaranteed” races

An abstraction of the program trace:

```html
<html><body>
<script>
var init = false, y = null;
function f() {
    if (init)
        alert(y.g);
    else
        alert("not ready");
}
</script>

<input type="button" id="bl" onclick="javascript:f()">

<script>
    y = { g:42 };
    init = true;
</script>
</body></html>
```

Common approach: record only shared reads and writes.
Wanted: “guaranteed” races

An abstraction of the program trace:

- `a: read(y)`
- `c: write(init)`
- `b: read(y.g)`
- `d: read(init)`

Which races are “guaranteed to exist”? Common approach: record only shared reads and writes.
Wanted: “guaranteed” races

An abstraction of the program trace:

```html
<html><body>
<script>
    var init = false, y = null;
    function f() {
        if (init)
            alert(y.g);
        else
            alert("not ready");
    }
</script>

<input type="button" id="b1" onclick="javascript:f()">

<script>
    y = { g:42 };
    init = true;
</script>

</body></html>
```
Wanted: “guaranteed” races

An abstraction of the program trace:

```html
<html><body>
<script>
var init = false, y = null;
function f() {
    if (init)
        alert(y.g);
    else
        alert("not ready");
}
</script>

<input type="button" id="b1" onclick="javascript:f()">

<script>
y = { g:42 };
init = true;
</script>

</body></html>
```
Wanted: “guaranteed” races

Which races are “guaranteed to exist”?

An abstraction of the program trace:

```html
<html><body>
<script>
    var init = false, y = null;
    function f() {
        if (init)
            alert(y.g);
        else
            alert("not ready");
    }
</script>

<input type="button" id="b1" onclick="javascript:f()">

<script>
    y = { g:42 };
    init = true;
</script>
</body></html>
```
Wanted: “guaranteed” races

(c,d) is a guaranteed race if \( \forall \pi' \in \gamma(\pi) \), c and d are a guaranteed race in the concrete trace \( \pi' \)

But how are we going to compute these in the abstract?
Key Idea 1: Coverage

(c,d) is a guaranteed race if \( \forall \pi' \in \gamma(\pi) \), c and d are a guaranteed race in the concrete trace \( \pi' \).

**Definition of <R>**

**Theorem:**

A <R> race is a guaranteed race.
Key Idea 1: Coverage

(c,d) is a guaranteed race if $\forall \pi' \in \gamma(\pi)$, c and d are a guaranteed race in the concrete trace $\pi'$

Definition: race $(c,d)$ covers race $(a,b)$ if $a \preceq c$ (or $a$ and $c$ are in the same action), and $d \preceq b$.

Generalizes to coverage by multiple races

Theorem:

An uncovered race is a guaranteed race.
Race Detection for Web: Challenges

- **Precision:** state-of-the-art detectors lead to too many *false positives*
  - caused by synchronization with read/writes, very common on the Web

- **Scalability:** state-of-the-art race detectors do not scale
  - blow-up in size of data structures caused by too many event handlers
Race Detection for Web: Challenges

- **Precision**: state-of-the-art detectors lead to too many false positives
  - caused by synchronization with read/writes, very common on the Web

- **Scalability**: state-of-the-art race detectors do not scale
  - blow-up in size of data structures caused by too many event handlers
Computing Races

A race detector should compute races. The basic query is whether two operations $a$ and $b$ are ordered:

$$a \preceq b$$

**Observation:** represent $\preceq$ as a directed acyclic graph and perform graph connectivity queries to answer $a \preceq b$. 
The happens-before graph

For this graph:

\[
\begin{align*}
A & \preceq B \\
A & \preceq C \\
B & \preceq D \\
C & \preceq D \\
D & \preceq E \\
A & \preceq D \\
A & \preceq E \\
C & \preceq E \\
B & \preceq E \\
B & \preceq C \\
\end{align*}
\]
\( a \preceq b \) via BFS

- Number of edges: \( M \)
- Number of nodes: \( N \)

Query Time: \( O(M) \)
Space: \( O(N) \)
a ≼ b via vector clocks
(classic race detection)

A vector clock vc is a map:
vc ∈ TID → N
associate a vector clock with each node

<1,0,0,0,0> ⊑ <1,1,1,1,0>
it follows that A ≼ D

<1,1,0,0,0> ≲ <1,0,1,0,0>
it follows that B ≲ C
a \leq b \text{ via vector clocks}

(\text{classic race detection})

\begin{align*}
\langle 1,0,0,0,0 \rangle & \rightarrow A \\
\langle 1,1,0,0,0 \rangle & \rightarrow B \\
\langle 1,1,1,0,0 \rangle & \rightarrow D \\
\langle 1,1,1,1,0 \rangle & \rightarrow E \\
\langle 1,1,1,1,1 \rangle & \rightarrow C \\
\end{align*}

\begin{align*}
\text{Pre-computation Time: } & O(M \cdot N) \\
\text{Query Time: } & O(1) \\
\text{Space: } & O(N^2)
\end{align*}

\text{Space Explosion}
Key idea: Re-discover threads by partitioning the nodes into chains.

due to:

“A Compression Technique to Materialize Transitive Closure”, 1990, H.V. Jagadish

computes a map:

$c \in \text{Nodes} \rightarrow \text{ChainIDs}$

associate a chain with each node
a \leq b \text{ via combining chain decomposition with vector clocks}

Key idea: Re-discover threads by partitioning the nodes into chains.

due to:

“A Compression Technique to Materialize Transitive Closure”, 1990, H.V. Jagadish

computes a map:

\[ c \in \text{Nodes} \rightarrow \text{ChainIDs} \]

associate a chain with each node
a \preceq b \text{ via combining chain decomposition with vector clocks (optimal version)}

C = \text{number of chains}

\text{Chain Computation Time: } O(N^3 + C \cdot M)

\text{Vector clock computation: } O(C \cdot M)

\text{Query Time: } O(1)

\text{Space: } O(C \cdot N)

Improved
\[ a \preceq b \text{ via combining chain decomposition with vector clocks} \]

(greedy version)

C = number of chains

Chain Computation Time: \( O(C \cdot M) \)

Vector clock computation: \( O(C \cdot M) \)

Query Time: \( O(1) \)

Space: \( O(C \cdot N) \)
Race Detection: Web

Race Detector

- Race Coverage
- Vector Clocks
- Chain Decomposition

Pre-computation Time: $O(C \cdot M)$
Query Time: $O(1)$
Space: $O(C \cdot N)$
Implementation

- Based on WebKit Browser
  - Used by Apple’s Safari and Google’s Chrome

- Quite robust, Demo:
  - [http://www.eventracer.org](http://www.eventracer.org)
Experiments: Fortune 100 web sites

Race Detector

- Happens-Before Model
- Memory Locations
- Race coverage
- Vector clocks
- Chain decomposition

~17 per web site
## Experiments: usability

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean # race vars</th>
<th>Max # race vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>634.6</td>
<td>3460</td>
</tr>
<tr>
<td>Only uncovered races</td>
<td>45.3</td>
<td>331</td>
</tr>
</tbody>
</table>

### Filtering methods

<table>
<thead>
<tr>
<th>Filtering methods</th>
<th>Mean # race vars</th>
<th>Max # race vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing same value</td>
<td>0.75</td>
<td>12</td>
</tr>
<tr>
<td>Only local reads</td>
<td>3.42</td>
<td>43</td>
</tr>
<tr>
<td>Late attachment of event handler</td>
<td>16.7</td>
<td>117</td>
</tr>
<tr>
<td>Lazy initialization</td>
<td>4.3</td>
<td>61</td>
</tr>
<tr>
<td>Commuting operations - className, cookie</td>
<td>4.0</td>
<td>80</td>
</tr>
<tr>
<td>Race with unload</td>
<td>1.1</td>
<td>33</td>
</tr>
<tr>
<td><strong>Remaining after all filters</strong></td>
<td><strong>17.8</strong></td>
<td><strong>261</strong></td>
</tr>
</tbody>
</table>
## Experiments: speed

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of event actions</td>
<td>5868</td>
<td>114900</td>
</tr>
<tr>
<td>Number of chains</td>
<td>175</td>
<td>792</td>
</tr>
<tr>
<td>Graph connectivity algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector clocks w/o chain decomposition</td>
<td>&gt;0.1sec</td>
<td>OOM</td>
</tr>
<tr>
<td>Vector clocks + chain decomposition</td>
<td>0.04sec</td>
<td>2.4sec</td>
</tr>
<tr>
<td>Breadth-first search</td>
<td>&gt;22sec</td>
<td>TIMEOUT</td>
</tr>
</tbody>
</table>
## Experiments: space

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of event actions</td>
<td>5868</td>
<td>114900</td>
</tr>
<tr>
<td>Number of chains</td>
<td>175</td>
<td>792</td>
</tr>
<tr>
<td>Graph connectivity algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector clocks w/o chain decomposition</td>
<td>544MB</td>
<td>25181MB</td>
</tr>
<tr>
<td>Vector clocks + chain decomposition</td>
<td>5MB</td>
<td>171MB</td>
</tr>
</tbody>
</table>
Manual inspection of 314 races

- **57%** are synchronization races
  - many idioms: conditionals, try-catch, looping over arrays

- **24%** are harmful races
  - many cases of reading from undefined
  - new bugs: UI glitches, broken functionality after a race, needs page refresh, missing event handlers, broken analytics.

- **17%** are harmless races
Future Work

- Race Detection as Abstract Interpretation
- Generalized Race Detection to Commutativity
- Synthesis of Repairs
- Reachability algorithms based on graph contraction
  - inspired by algorithms for road networks
- Stateless model checking
  - race-guided exploration of the web page
Summary

- Introduced Happens-Before model for web applications
  - useful for any concurrency analysis
- Race coverage: report only real races
- Efficient Analysis
  - combines vector clocks, chain decomposition and race coverage
Try it out

http://www.eventtracer.org