Data-Race Exceptions Have Benefits Beyond the Memory Model

Benjamin P. Wood, Luis Ceze, Dan Grossman

University of Washington
Why data-race exceptions?

[Elmas et al., PLDI 2007; Adve and Boehm, CACM Aug. 2010; Marino et al., PLDI 2010; Lucia et al., ISCA 2010; ...]
Why data-race exceptions?

Find bugs.

[Elmas et al., PLDI 2007; Adve and Boehm, CACM Aug. 2010; Marino et al., PLDI 2010; Lucia et al., ISCA 2010; ...]
Why data-race exceptions?

Find bugs.

Simplify memory models.

(DRF $\Rightarrow$ SC)

[Elmas et al., PLDI 2007; Adve and Boehm, CACM Aug. 2010; Marino et al., PLDI 2010; Lucia et al., ISCA 2010; ...]
Why data-race exceptions?

- Find bugs.
- Simplify memory models.
- Avoid reasoning about memory reorderings.

[Elmas et al., PLDI 2007; Adve and Boehm, CACM Aug. 2010; Marino et al., PLDI 2010; Lucia et al., ISCA 2010; ...]
Why *not* data-race exceptions?
Why *not* data-race exceptions?

Lock-free algorithms
Why *not* data-race exceptions?

“Benign” races

Lock-free algorithms
Why *not* data-race exceptions?

“Benign” races

Lock-free algorithms

*unchecked annotations*
Why *not* data-race exceptions?

- "Benign" races
- Lock-free algorithms
- *unchecked* annotations

Performance overheads

Ongoing research
Overheads

- costs

Find bugs.
Simplify memory models.

benefits
Find bugs. Simplify memory models.

Overheads

costs

benefits
Overheads

costs

Find bugs. Simplify memory models.

benefits
This talk explores hidden benefits of data-race exceptions in runtime systems.
This talk explores hidden benefits (and costs) of data-race exceptions \textbf{in runtime systems}. 
This talk explores hidden benefits (and costs) of data-race exceptions in runtime systems.
All races are inherently wrong.
All races are inherently wrong.
Attempts at racy accesses are exceptional, but legal.

Exceptions ensure all races are impossible.

All races are inherently wrong.
Contributions

Properties of data-race exceptions enable **conflict detection**.

Exploit data-race exceptions in **concurrent garbage collection**.

Low-level data-race exceptions have **subtle implications**.
outline

Review data races, exceptions, and sequential consistency.

Properties of data-race exceptions enable conflict detection.

Exploit data-race exceptions in concurrent garbage collection.

Low-level data-race exceptions have subtle implications.

Conclusions
**data race**

A pair of *concurrent, conflicting* accesses

Thread 1

\[
x := 1 \\
\text{release}(m) \\
r1 := y
\]

Thread 2

\[
x := 1 \\
\text{acquire}(m) \\
r2 := x \\
y := 1
\]

---

Synchronization
Thread 1

\[ x := 1 \]
\[ \text{release}(m) \]
\[ r1 := y \]

Thread 2

\[ \text{synchronization} \]
\[ \text{acquire}(m) \]
\[ r2 := x \]

Data race

\[ y := 1 \]

Exception on second access
data-race exceptions

 guarantee either data-race-free or exception

Thread 1

\[
\begin{align*}
  x &:= 1 \\
  \text{release}(m) \\
  r1 &:= y
\end{align*}
\]

exception on second access

Thread 2

\[
\begin{align*}
  \text{acquire}(m) \\
  r2 &:= x \\
  y &:= 1
\end{align*}
\]
data-race exceptions

guarantee either data-race-free or exception

Thread 1

\[ x := 1 \]

\[ \text{release}(m) \]

\[ r1 := y \]

Thread 2

\[ \text{acquire}(m) \]

\[ r2 := x \]

exception on second access
Data-race exceptions

DRF $\oplus$ exception and

Java/C++

$\rightarrow$

SC or exception
data-race exceptions in HW

precise
Suspend the thread just before its racy access. Respect program order.

handleable
Deliver a trap with information about the race.

```
acquire(m)

r2 := x
y := 1
r3 := z
```

exception delivery
concurrent GC

**Concurrent mark-sweep:** atomic/consistent heap traversal

*tri-color marking and write barriers*

**Concurrent copying/moving:** atomic object copying

Piggyback on STM runtime [McGachey et al., PPoPP 2008]

Lock-free algorithms [Pizlo et al., ISMM 2007, PLDI 2008]
mark-sweep

GC thread  Mutator

gray(a)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
mark-sweep

**GC thread**  **Mutator**

gray(a)

gray(a.next)

black(a)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)
mark-sweep

**GC thread**  Mutator

- **gray(a)**
- **gray(a.next)**
- **black(a)**

```plaintext
n := a.next
a.next := a.next.next
n.next := null
```

![Heap Diagram]

- **black**: reachable, refs visited
- **dark gray**: reachable, refs unvisited
- **white**: unreachable/unknown
mark-sweep

**GC thread**  Mutator

gray(a)
gray(a.next)
black(a)

\[ n := a.next \]
\[ a.next := a.next.next \]
\[ n.next := \text{null} \]

gray(b.next)
black(b)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)

\[
\begin{align*}
n &:= a.\text{next} \\
a.\text{next} &:= a.\text{next}.\text{next} \\
n.\text{next} &:= \text{null}
\end{align*}
\]
gray(b.next)
black(b)
mark-sweep

**GC thread**  **Mutator**

- gray(a)
- gray(a.next)
- black(a)
- n := a.next
- a.next := a.next.next
- n.next := null

- gray(b.next)
- black(b)

- collect(c)
mark-sweep

**GC thread**  Mutator

gray(a)
gray(a.next)
black(a)

\[ n := a.\text{next} \]
\[ a.\text{next} := a.\text{next}.\text{next} \]
\[ n.\text{next} := \text{null} \]

gray(b.next)
black(b)

collect(c)
mark-sweep

**GC thread**  Mutator

gray(a)
gray(a.next)
black(a)  \[ n ::= a.next \]
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)

\[ n := a.next \]
\[ a.next := a.next.next \]
gray(c)

Heap

reachable, refs visited
reachable, refs unvisited
unreachable/unknown
**mark-sweep**

**GC thread**  Mutator

gray(a)

gray(a.next)

black(a)

\[
n := a.\text{next}
\]

\[
a.\text{next} := a.\text{next}.\text{next}
\]

gray(c)

---

**Heap**

- reachable, refs visited
- reachable, refs unvisited
- unreachable/unknown
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)

n := a.next
a.next := a.next.next

gray(c)
n.next := null

gray(b.next)
black(b)
mark-sweep

GC thread   Mutator

\[
gray(a) \\
gray(a.next) \\
black(a) \\
\]

\[
n := a.next \\
a.next := a.next.next \\
gray(c) \\
n.next := \text{null} \\
\]

\[
gray(b.next) \\
black(b) \\
\]

Heap

- reachable, refs visited
- reachable, refs unvisited
- unreachable/unknown
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)

n := a.next
a.next := a.next.next
gray(c)
n.next := null

gray(b.next)
black(b)
gray(c.next)
black(c)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)
n := a.next
a.next := a.next.next
gray(c)
n.next := null

gray(b.next)
black(b)
gray(c.next)
black(c)
mark-sweep

GC thread  Mutator

gray(a)
gray(a.next)
black(a)

\[ n := a.\text{next} \]
\[ a.\text{next} := a.\text{next}.\text{next} \]
gray(c)
\[ n.\text{next} := \text{null} \]
gray(b.next)
black(b)
gray(c.next)
black(c)

Heap

- reachable, refs visited
- reachable, refs unvisited
- unreachable/unknown
mark-sweep

GC thread
gray(a)
gray(a.next)
black(a)

Mutator
n := a.next
a.next := a.next.next
gray(c)
n.next := null

Heap

a
black
b
gray(b.next)
black(b)
gray(c.next)
black(c)

exception: gray(c)

reachable, refs visited
reachable, refs unvisited
unreachable/unknown
mark-sweep

**GC thread**
gray(a)
gray(a.next)
black(a)

**Mutator**

\[
n := a.next
\]

\[
a.next := a.next.next
\]

gray(c)

\[
n.next := \text{null}
\]

gray(b.next)

black(b)

gray(c.next)

black(c)

**Heap**

- reachable, refs visited
- reachable, refs unvisited
- unreachable/unknown

**Exception:**

gray(c)

gray(c)

exception: ???
extensions for GC

Erase tracks
Remove and replace last reads and last writes.

Racy read
Record and execute a read even if it races.

Data-carrying exceptions
Deliver a racing write value to a racing read.

...
SO FAR

Use HW data-race exceptions in runtime systems.
Use HW data-race exceptions in runtime systems.

Do guarantees from HW data-race exceptions apply at the program level?
high-level support
non-program accesses and sync.
caveats for any GC

Movement
Race-detector state must follow moved objects atomically.

Invisibility
Only program heap writes should be seen by the race detector. GC writes to the heap should be ignored.

No transitive ordering
GC must not induce ordering between mutators.
cooperation vs. abstraction
also in the paper...

More details on concurrent GC using data-race exceptions

Lock elision with no rollback support

Conflict races and conflict-race exceptions
conclusions

Data-race exceptions have benefits beyond the memory model.

Exceptions make races impossible. Attempted races are exceptional, but legal.

Data-race exceptions enable general conflict detection for runtime systems like concurrent GC.

Low-level data-race detection has subtle implications.
This slide intentionally not left blank.
conflict race

data race across synchronization-free regions running concurrently in real time

Thread 1

wr x

rd x

rd y

Thread 2

wr y

rd x

...
Thread 1
- wr x
- rd x
- rd y

Thread 2
- wr y
- rd x

Data race, no conflict race
**conflict-race exceptions**

guarantee sequential consistency or exception

Thread 1

- `wr x`
- `rd x`
- `rd y`

Thread 2

- `...`
- `wr y`
- `rd x`

Data race, no conflict race
conflict-race exceptions
guarantee sequential consistency or exception
Best-effort automatic recovery from conflict races

Thread 1
- wr x
- wr y
- rd y

Thread 2
- ... (omitted)
- rd x
- [delay]
- rd x

(time)

1
2
Best-effort automatic recovery from conflict races

Thread 1
- wr x
- wr y
- rd y

Thread 2
- ... [delay]
- rd x

1 exception

Time
Best-effort automatic recovery from conflict races

Thread 1

- wr x
- wr y
- rd y

Thread 2

- ... (race)
- rd x
- [delay]

- 1 exception
- 2 success
moving/copying

GC Thread         Mutator         Resulting Heap

0

p = o.forward
p.x = 1

forward
x 1

o' = malloc(...)
o'.forward = o'

0
moving/copying

GC Thread | Mutator | Resulting Heap
---|---|---
0 | p = o.forward
p.x = 1 | o.forward = o'
1 | o' = malloc(...) | o'.forward = o'

```
tmp = o.x
o'.x = tmp
o.forward = o'
p = o.forward
p.x = 2
p = o.forward
p.x = 1
```
moving/copying

**GC Thread**

0

p = o.forward
p.x = 1

1

o' = malloc(...)
o'.forward = o'

2

tmp = o.x
o'.x = tmp

**Mutator**

**Resulting Heap**
moving/copying

<table>
<thead>
<tr>
<th>GC Thread</th>
<th>Mutator</th>
<th>Resulting Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p = o.forward &lt;br&gt;p.x = 1</td>
<td>![Resulting Heap 0]</td>
</tr>
<tr>
<td>1</td>
<td>o' = malloc(...) &lt;br&gt;o'.forward = o'</td>
<td>![Resulting Heap 1]</td>
</tr>
<tr>
<td>2</td>
<td>tmp = o.x &lt;br&gt;o'.x = tmp</td>
<td>![Resulting Heap 2]</td>
</tr>
<tr>
<td>3</td>
<td><strong>R-W race</strong> &lt;br&gt;p = o.forward &lt;br&gt;p.x = 2</td>
<td>![Resulting Heap 3]</td>
</tr>
</tbody>
</table>
moving/copying

**GC Thread**

1. `p = o.forward`
   - `p.x = 1`

2. `o' = malloc(...)`
   - `o'.forward = o'`

3. `tmp = o.x`
   - `o'.x = tmp`

4. `R-W race`
   - `p = o.forward`
   - `p.x = 2`

5. `o.forward = o'`

**Mutator**

- `p = o.forward`
- `p.x = 1`
- `o'.forward = o'`
- `o'.x = 1`
- `R-W race`
- `p = o.forward`
- `p.x = 2`
- `o.forward = o'`