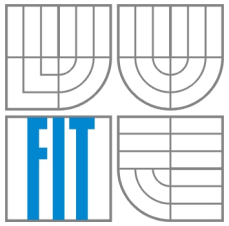


Healing Data Races On-The-Fly

Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar
(Brno university of Technology, Czech Republic)

Rachel Tzoref, Shmuel Ur
(IBM, Haifa Research Lab, Israel)

Agenda

- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

What is a race?

- A data race occurs when two concurrent threads access a shared variable and when:
  - at least one access is a write and
  - the threads use no explicit mechanism to prevent the accesses from being simultaneous.

- Usually a data race is a serious error caused by failure to synchronize properly.
- Can cause wrong results, deadlocks, exceptions...

Atomicity races

- Races caused by violation of wrong assumptions that some blocks of code will be executed atomically.
- Example:

Thread 1

```
void someMethod(){
   long local = shared;
   local = update(local);
   shared = local;
}
```

Thread 2

```
void someMethod(){
   long local = shared;
   local = update(local);
   shared = local;
}
```

Atomicity races

- Races caused by violation of wrong assumptions that some blocks of code will be executed atomically.
- Example:

Thread 1

```
void someMethod(){
   shared=update(shared);
}
```

Thread 2

```
void someMethod(){
   shared=update(shared);
}
```

Inherent races

- Races not related to atomicity.
- Data race if the following holds:
  - Executing any segment of code in each thread atomically does not determine an order of accesses to shared variable.
  - The different orders in which the shared variable is acessed can be classified as “good” and “bad” according to the expected behaviour of the program.

Inherent races

- Example:

Thread 1

```
void synchronized
            someMethod(){
   long local = shared;
   local = update(local);
   shared = local;
}
```

Thread2

```
   void synchronized
            otherMethod(){
      shared = null;
   }
```

Agenda

- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

Problem detection

- Eraser algorithm
  - Detects so called apparent data races
- Principle:
  - For each variable maintains its state and the set of candidate locks
  - Race is detected whenever:
    - the variable is in state Shared and
    - the set of candidates locks becomes empty

Demonstration of the detection

```
static class Flight {
   private int soldSeats;
   ...
   Flight(){
      soldSeats = 0;
      ...
   }
   ...
   boolean bookTicket(){
      soldSeats++;
   }
   ...
}
```

Thread_main

```
new Flight();
```

(state = Exclusive, C(v)={})

Thread 1

```
synchronized(lock){
bookTicket();}
```

(state = Shared, C(v)={lock})

Thread 2

```
bookTicket();
```

(state = Race, C(v)={})

Time

Agenda

- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

Problem localisation

- Often hard task even for humans.
- Oracle based on looking for pre-specified data race bug patterns in the code with the aid of information collected by race detector.
- Use formal methods to reduce the number of false alarms but with reasonable overhead.

Atomicity Violation Bug Patterns

- Load-store bug pattern

```
x++;
```

```
2: getfield     #2
5: iconst_1
6: iadd
7: putfield     #2
```

- Test-and-use bug pattern

```
if (p != null)
   p = p.next;
```

```
0: aload_0
1: getfield     #2
4: ifnull    18
7: aload_0
8: aload_0
9: getfield     #2
12:getfield     #3
15:putfield     #2
18:...
```

- Repeated test-and-use pattern

```
while (p != null)
   p = p.next;
```

Demonstration of the localisation

```
static class Flight {
   private int soldSeats;
   ...
   Flight(){
      soldSeats = 0;
      ...
   }
   ...
   boolean bookTicket(){
      soldSeats++;
   }
   ...
}
```

```
getfield    #2
iconst_1
iadd
putfield    #2
```

Agenda

- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

Healing atomicity races

- Influencing the scheduler
  - Forcing a context switch

```
Thread.yield();
```

- Idea:
  - To receive full time window from the scheduler.
- Pros
  - Safe and legal solution.
- Cons
  - Only decrease the probability of race manifestation.

Healing atomicity races

- Influencing scheduler
  - Temporary changes of the priorities

```
Thread.setPriority(MAXPRIORITY);
...
Thread.setPriority(originalPriority);
```

- Idea:
  - To receive full time window from the scheduler.
- Pros
  - Safe and legal solution.
- Cons
  - Only decrease the probability of race manifestation.
  - Strongly JVM and OS dependent.

Healing atomicity races

- Adding synchronization actions
  - Suitable use of mutexes

```
healingMutex.lock();
...
healingMutex.unlock();
```

- Idea:
  - To prevent accesses being simultaneous.
- Pros
  - Heal the race.
- Cons
  - Could introduce deadlock.
  - Exceptions must be covered.

Healing inherent races

- Distinguish between good and bad orders
- Enforce good order
  - Change the scheduling of the program.
- Override bad order
  - Concentrate on multiple write accesses.
  - Doesn't prevent bad order from occurring.

Demonstration of the healing

```
static class Flight {
   private int soldSeats;
   ...
   Flight(){
      soldSeats = 0;
      ...
   }
   ...
   boolean bookTicket(){
      soldSeats++;
   }
   ...
}
```

Healing by influencing scheduler:

```
boolean bookTicket(){
      Thread.yield();
      soldSeats++;
   }
```

Healing by synchronization:

```
boolean bookTicket(){
      raceLock.lock();
      soldSeats++;
      raceLock.unlock();
   }
```

Agenda

- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

Healing assurance

- Static analysis and/or bounded model checking
  - Reduce false alarms during detection and localisation.
  - Ensure that a new bug cannot be introduced.
  - Help to choose suitable healing method.

- ... future work

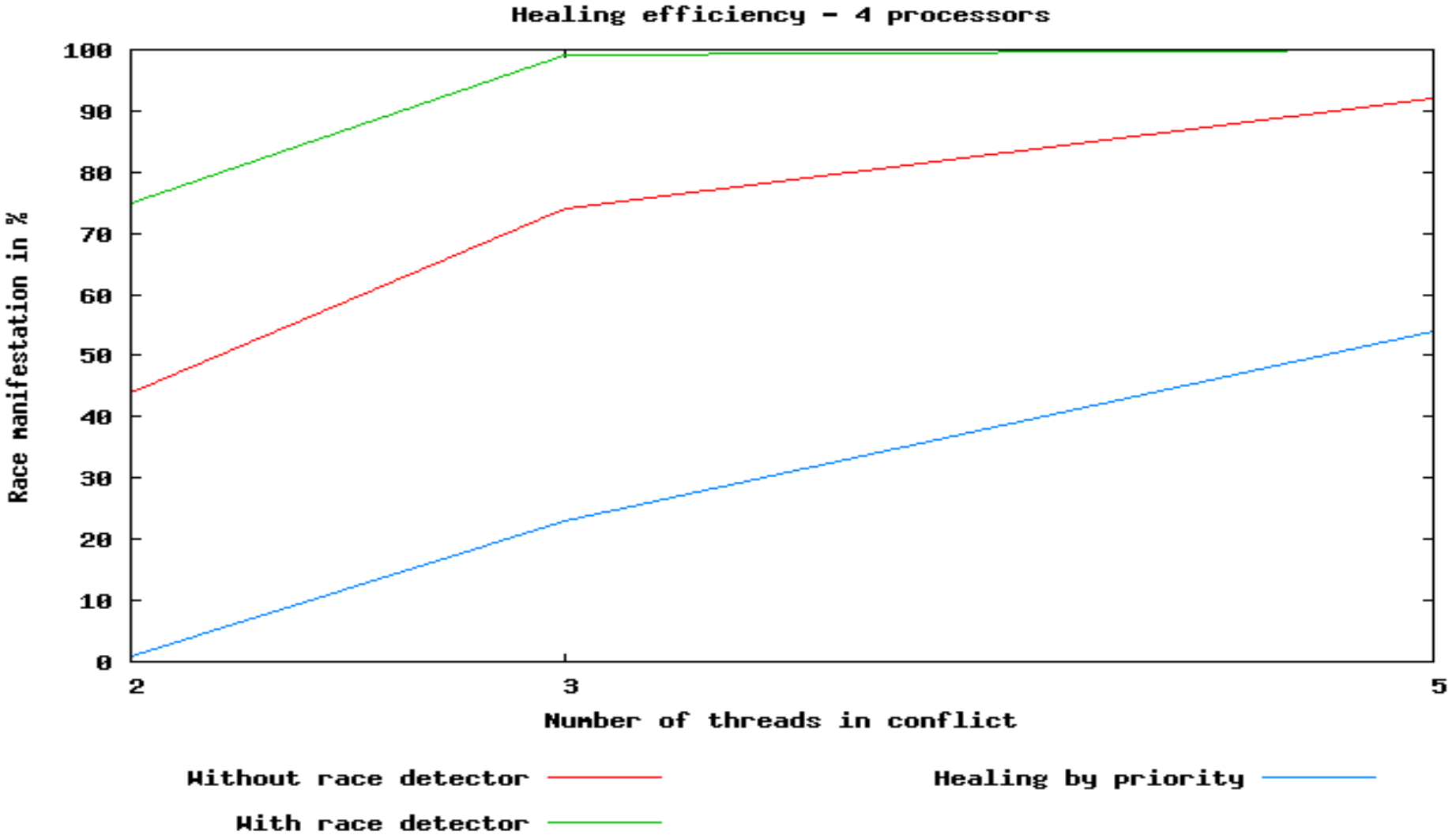
Agenda

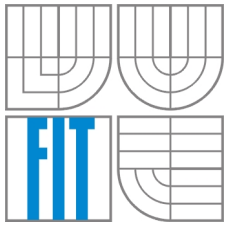
- Introduction
- Self-Healing steps:
  - Problem detection
  - Problem localisation
  - Problem healing
  - Healing assurance
- Preliminary results and experiments
- Discussion

Preliminary results

- Implemented race detector is able:
  - To detect wrong locking policy using Eraser algorithm.
  - To detect load-store atomicity bug pattern.
  - To localise the bug and give enough information to the developer.
  - To heal founded bug by influencing scheduler and also by introducing additional synchronization.
- Architecture available also as an Eclipse plug-in.

Experiments



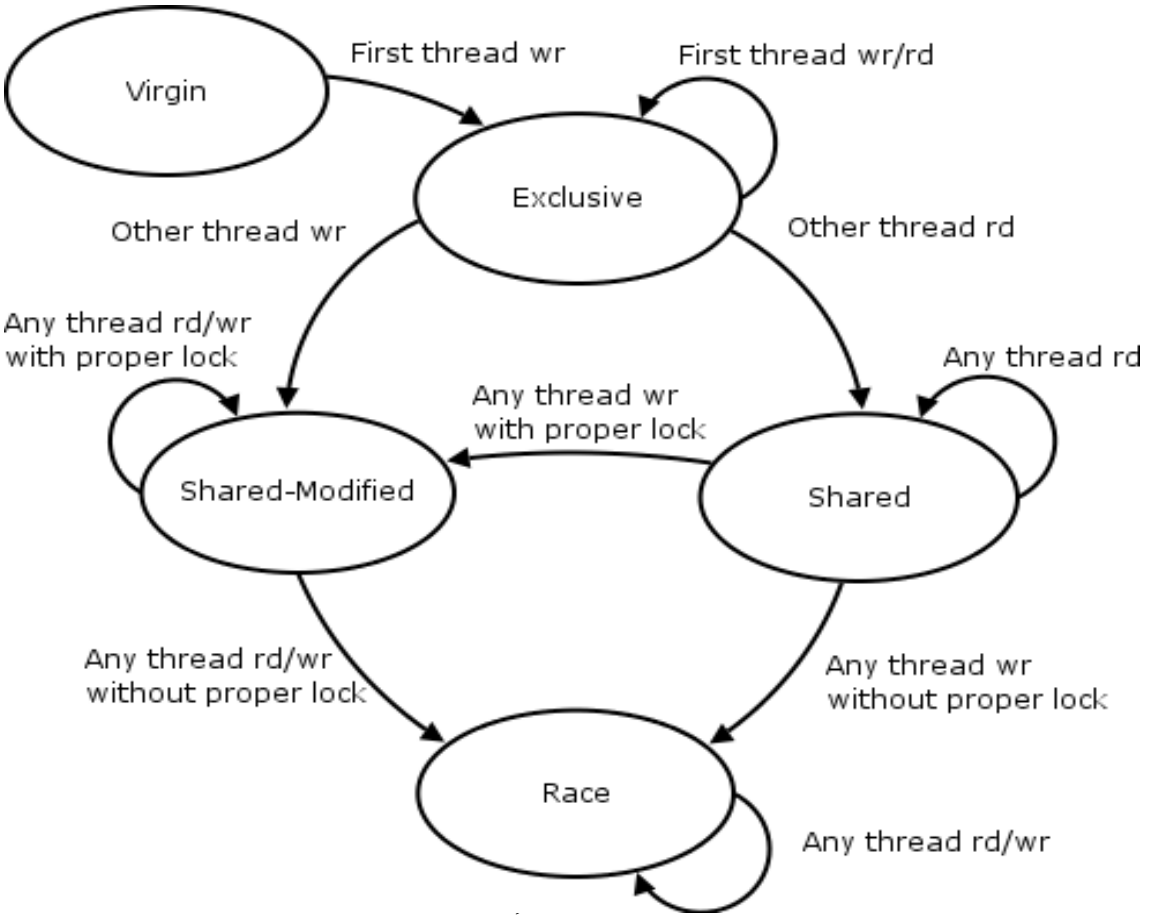


Thank you

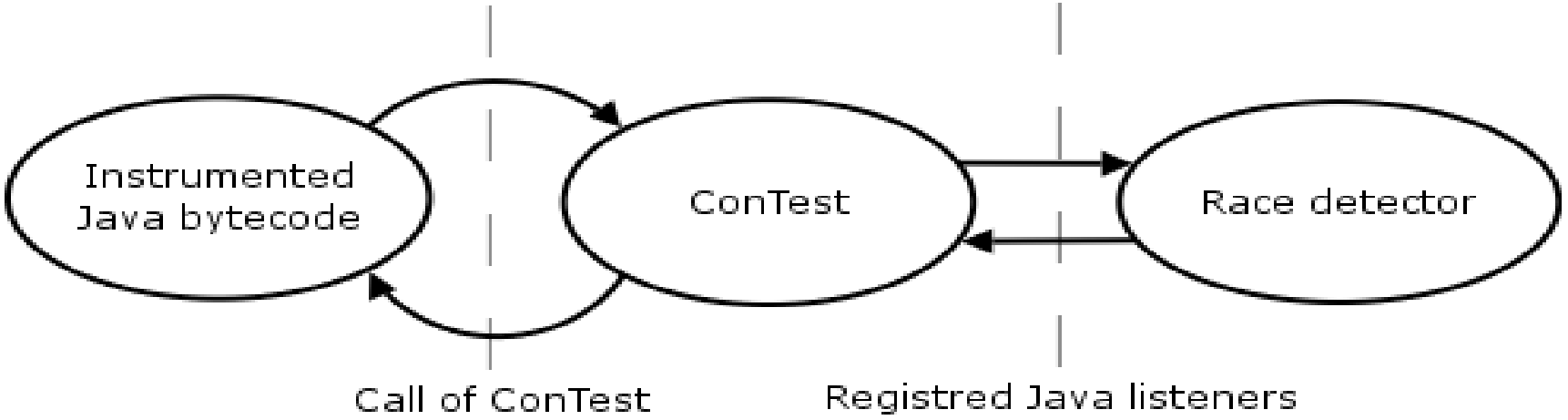
Eraser algorithm

Candidate_locks(x) := Candidate_locks(x) ∩ Locks_held(x);
if Candidate_locks(x) = { }, then issue a warning

Candidate_locks = set of locks used to protect variable
Locks_held = set of locks owned by thread



Architecture overview



Original bytecode

```
...
load x
...
```

Instrumented bytecode

```
...
BeforeVarRead
load x
AfterVarRead
...
```