University of Stuttgart
Institute of Software Technology

Eliminating Data Race Warnings Using CSP

Martin Wittiger

Ada-Europe 2016

The Problem with Data Races

Definition of Data Race

A program contains a data race if two concurrently running tasks access the same piece of memory, one of those accesses is a write, and there is no synchronization that guarantees the accesses are not simultaneous.

```
…
shared += 3;
…
```

Task A

```
…
shared -= 1;
…
```

Task B

The Problem with Data Races

Definition of Data Race

A program contains a data race if two concurrently running tasks access the same piece of memory, one of those accesses is a write, and there is no synchronization that guarantees the accesses are not simultaneous

Task A

```
…
r = shared;
r += 3;
shared = r;
…
```

10

.
.
.

Task B

```
…
r = shared;
r--;
shared = r;
…
```

9? 12? 13?

The Problem with Data Races

Implications

- C programs that contain data races have undefined behaviour
- Most data races “occur” only under rare timing conditions
  Scheduling is typically indeterministic: The erroneous behaviour may reveal itself in some executions but not in all
  - Data races cannot be found reliably by testing
  - In many cases the erroneous behaviour is often not reproducible
- Data races are a safety nightmare for embedded systems where failure may lead to loss of life

Step 1:
Static Data Race Analysis

Step 1: Static Data Race Analysis

Setup

- Idea: Use conservative static analysis to find all possible data races
  - Published by (amongst others) Vaziri et al. in 2006
  - Shown to work on industry-sized systems
- Our analysis relies on control flow, pointer, lockset and escape analysis (all of them sound in a concurrent setup)
- Input is C source code and configuration
- Output is a list of potential data race pairs, so-called data race warnings
- This list has to be assessed manually for it may (and in practice will) contain many false positive warnings

Step 1: Static Data Race Analysis

Manual Assessment of Data Race Warnings

When examining data race warnings, we look for reasons to exclude the data race

- Mutexes or Locks → typically not in embedded systems
- Interrupt disable/enable patterns → already taken into account by analysis
- System state or state-based synchronisation

Task A

```
if (state == RUN) {
    …
    shared += 3;

    …
}
```

DRW

Task B

```
if (state == SHUTDOWN) {
    …
    shared -= 1;

    …
}
```

Step 1: Static Data Race Analysis

System State

- Deciding whether the system state precludes data races is difficult
  - When does the state change?
  - Are we sure we are aware of all writes to the state variable?

  But maybe, if we stick to a simple pattern that works…

Step 2:
Static Data Race Analysis with State Pattern Recognition

Step 2: Static Data Race Analysis with State Pattern Recognition

Concept

- Idea: Write analysis code that recognizes a specific state machine pattern
- Published by Keul in 2011, others have implemented variations
- Clearly an improvement, reduces false positive warnings

```
typedef enum {INIT, RUN, SHUTDOWN} sys_state;
volatile sys_state state = INIT;
```

```
if (state == INIT) {
   // do something
   …
   state = RUN;
}
```

```
if (state == RUN) {
   // do something
   …
}
```

Step 2: Static Data Race Analysis with State Pattern Recognition

The Problem Persists

- But: Though this reduces the number of false positive warnings on some systems, when examining real systems the picture has not changed
  - Slight variations mean the pattern is no longer recognized
  - Consider:
    - Macro-Constants or Integer-Literals used instead of enums
    - Initialisation works slightly differently
    - Additional reads on the state variable
    - state variable has address taken
    - “aborting” assignments to variables
- Big question: Does the pattern still work?
- In practice: Variations are often not slight, but “creative”

Final Solution:
Static Data Race Analysis Using Refinement Checkers

Final Solution: Static Data Race Analysis using Refinement Checkers

Concept

- Refinement checkers exist for CSP
  - Communicating Sequential Processes
  - Developed by Hoare
  - Language to mathematically model concurrent processes
- Idea: Conservatively approximate system behaviour by projecting it on the effects on some state variables, then ask the refinement checker to prove the infeasibility of a specific data race situation

Final Solution: Static Data Race Analysis Using Refinement Checkers

Requirements on State Variables

- To be suitable for synchronization, state variables have to behave in a *sequentially consistent* manner and be *atomic*
- In short: We assume that this is the case when variables are declared as `volatile int`
  - Technically not in line with C99 Standard
  - Very reasonable assumption nonetheless
  - (The listener is referred to the paper for more details.)

Final Solution: Static Data Race Analysis Using Refinement Checkers

In brevity, steps performed:

- Control flow/pointer analysis to establish CFG/Points-To-Sets
- Escape analysis/Data-Race Analysis to establish DRWs
- Constant Propagation/Folding and state variable suitability check

- Manually select one or more state variables to be used
- Then, automatically transform (project) system to CSP, pre-process CSP and run the refinement checker
  We use FDR2 (Oxford University/Formal Systems Ltd.)

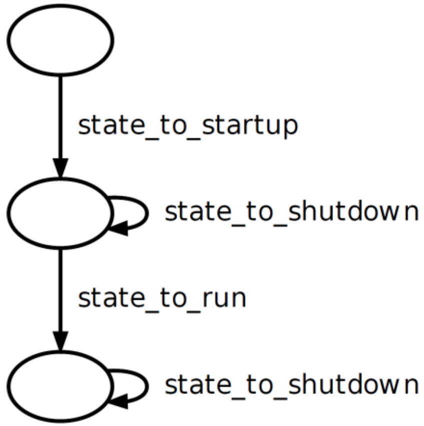
Final Solution: Static Data Race Analysis Using Refinement Checkers

Illustrative Examples

```
typedef enum {STARTUP, RUN, SHUTDOWN} State;
int x, y;
volatile State state;
```

```
void task_low (void) {
   state = STARTUP;
   x = 1;
   state = RUN;
   while (true)
      if (state == RUN)
         work ();
      else
         y++; }
```

```
void task_high (void) {
   if (state != STARTUP)
      x++;
   if (state != SHUTDOWN)
      y++;
   if (IND)
      state = SHUTDOWN;
}
```



The refinement checker refuses to eliminate the DRW!

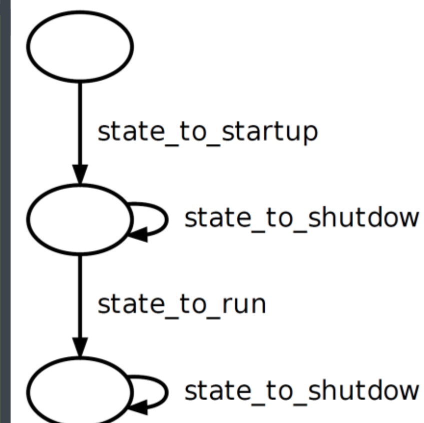
Final Solution: Static Data Race Analysis using Refinement Checkers

Illustrative Examples

```
typedef enum {STARTUP, RUN, SHUTDOWN} State;
int x, y;
volatile State state;
```

```
void task_low (void) {
   state = STARTUP;
   x = 1;
   state = RUN;
   while (true)
      if (state == RUN)
         work ();
      else
         y++; }
```

```
void task_high (void) {
   if (state != STARTUP)
      x++;
   if (state != SHUTDOWN)
      y++;
   if (IND)
      state = SHUTDOWN;
}
```



The refinement checker does eliminate the DRW!

Final Solution: Static Data Race Analysis Using Refinement Checkers

Practicality

- Data race analysis only needs to be run once
- Analysis steps other than refinement checker per warning ~30 s runtime
  - Highly parallelizable, analysing 40 warnings also takes ~30 s on multicore machine
  - Slightly revised approach runs in ~1 s
- Rule of thumb for refinement checker:
  If there is no result after 5 seconds, there is likely never going to be one.
- Has ruled out actual DRW on industry-sized systems
  by recognizing an intricate and complicated state-based synchronization scheme
- But, deviating from simple patterns very often breaks synchronization properties

Future Work

Future Work

- Working on further automisation,
  i. e. (heuristically) advising the user which state variables to pick/not to pick
- Improving speed, ratio of successful terminations on more state variables
- Interactive operation modes? Use results as visualisation?

University of Stuttgart
Institute of Software Technology

Questions?

Martin Wittiger

e-mail martin.wittiger@informatik.uni-stuttgart.de
phone +49 (0) 711 685-882-84
fax +49 (0) 711 685-883-80

University of Stuttgart
Institute of Software Technology
Universitätsstr. 38, 70569 Stuttgart