Effective Data-Race Detection for the Kernel

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk

Microsoft Research

Symposium on Operating Systems Design and Implementation (OSDI), October 2010

Motivation

- Finding data races is hard
- Analysing them is even harder
- Races often indicate problems
- Kernel code “operates at a lower concurrency abstraction” than user code

Motivation

- Finding data races is hard
- Analysing them is even harder
- Races often indicate problems
- Kernel code “operates at a lower concurrency abstraction” than user code

```
struct{
  int status:4;
  int pktRcvd:28;
} st;
```

Thread 1

```
st.status = 1;
```

Thread 2

```
st.pktRcvd ++;
```

Data Races

- *Two operations that access main memory are called conflicting if*
  - *the physical memory they access is not disjoint,*
  - *at least one of them is a write, and*
  - *they are not both synchronization accesses.*
- *A program has a data race if it can be executed on a multiprocessor in such a way that two conflicting memory accesses are performed simultaneously (by processors or any other device).*

Race Detection

Precision

Missed race	No warning from detector
Benign race	Race without negative effects on program behaviour
False race	Error reported even though there is no race

Race Detection

Precision

Missed race No warning from detector

Benign race Race without negative effects on program behaviour

False race Error reported even though there is no race

Detection Techniques

Static

Analyse source or byte code

Dynamic

Instrument program and monitor execution

Race Detection

Precision

Missed race No warning from detector

Benign race Race without negative effects on program behaviour

False race Error reported even though there is no race

Detection Techniques

Static

Analyse source or byte code

Dynamic

Happens-Before-Tracking

Record ordering of events and synchronisation operations

Lock Sets

Examine lock set (held locks) during each data access

Data Collider

- Detects data races in existing Windows kernel code (x86)
- Independent of synchronisation protocols
- Extra debugging information about the race (stack trace, “context information”)
- Runtime overhead below $5\,\%$ due to sampling
- Post-processing to prune and prioritise found races

Sampling Algorithm

1. Identify instructions that access data
2. Prune synchronisation instructions (`volatile`, hardware synchronisation instructions)
3. Choose breakpoints uniformly from sampling set initially and after race detection
4. Periodically readjust according to number of fired breakpoints per second

$\Rightarrow$ Effective at low sampling rates

Algorithm

```
AtPeriodicIntervals() {
  // determine k based on desired
  // memory access sampling rate
  repeat k times {
    pc = RandomlyChosenMemoryAccess();
    SetCodeBreakpoint(pc);
  }
}
```

```
OnCodeBreakpoint(pc) {
  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm(pc);

  DetectConflicts(loc, size, isWrite);

  // set another code break point
  pc = RandomlyChosenMemoryAccess();
  SetCodeBreakpoint(pc);
}
```

Algorithm

```
DetectConflicts(loc, size, isWrite) {
  temp = read(loc, size);

  if (isWrite) {
    SetDataBreakpointRW(loc, size);
  } else {
    SetDataBreakpointW(loc, size);
  }

  delay();

  ClearDataBreakpoint(loc, size);
  temp' = read(loc, size);

  if (temp != temp' || data breakpoint fired) {
    ReportDataRace( );
  }
}
```

Data Race Detection

Hardware Data Breakpoints (of x86)

- Based on virtual addresses
- IPI to update atomically on all cores
- Write $\rightarrow$ Trap on read/write
- Read $\rightarrow$ Trap on write

Data Race Detection

Hardware Data Breakpoints (of x86)

- Based on virtual addresses
- IPI to update atomically on all cores
- Write $\rightarrow$ Trap on read/write
- Read $\rightarrow$ Trap on write

Repeated Reads

- No detection of conflicting reads or writes with same last value
- Detect concurrent DMA writes
- Fallback when out of hardware data breakpoint
- Workaround for different virtual addresses mapping to same physical address

Pruning Benign Races

Statistics Counters	Counters that maintain low-fidelity statistical data
Safe Flag Updates	Read a bit while a different bit is updated
Special Variables	Races are expected, e.g. current time

$\Rightarrow$ $\sim$90 % of detected data races are benign

$\Rightarrow$ Still reported but deprioritised

Evaluation — Effectiveness

- *[A]pplied DataCollider on several modules in the Windows operating system [...] class drivers, various PnP drivers, local and remote file system drivers, storage drivers, and the core kernel executive itself*
- *[B]enign data races pruned heuristically and manually*

Evaluation — Effectiveness

- *[A]pplied DataCollider on several modules in the Windows operating system [...] class drivers, various PnP drivers, local and remote file system drivers, storage drivers, and the core kernel executive itself*
- *[B]enign data races pruned heuristically and manually*

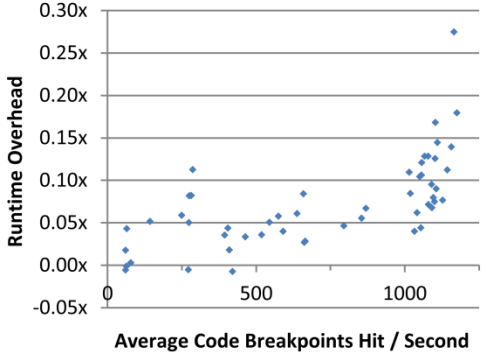
Data Races Reported	Count
Fixed	12
Confirmed and Being Fixed	13
Under Investigation	8
Harmless	5
Total	38

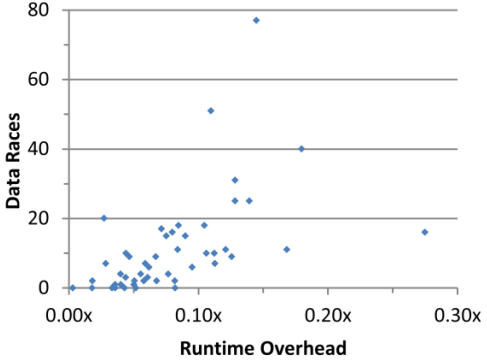
Evaluation — Overhead

[W]e repeatedly measured the time taken for the boot–shutdown sequence for different sampling rates and compared against a baseline Windows kernel running without DataCollider. These experiments where done on the x86 version of Windows 7 running on a virtual machine with 2 processors and 512 MB memory. The host machine is an Intel Core2-Quad 2.4 GHz machine with 4 GB memory running Windows Server 2008.

Evaluation — Overhead

[W]e repeatedly measured the time taken for the boot–shutdown sequence for different sampling rates and compared against a baseline Windows kernel running without DataCollider. These experiments where done on the x86 version of Windows 7 running on a virtual machine with 2 processors and 512 MB memory. The host machine is an Intel Core2-Quad 2.4 GHz machine with 4 GB memory running Windows Server 2008.





Evaluation — Efficacy of Pruning

We enabled DataCollider while running kernel stress tests for 2 hours sampling at approximately 1000 code breakpoints per second.

Evaluation — Efficacy of Pruning

We enabled DataCollider while running kernel stress tests for 2 hours sampling at approximately 1000 code breakpoints per second.

Data Race Category		Count
Benign — Heuristically Pruned	Statistic Counter	52
	Safe Flag Update	29
	Special Variable	5
Benign — Manually Pruned	Double-check locking	8
	Volatile	8
	Write Same Value	1
	Other	1
Real	Confirmed	5
	Investigating	4
Total		113

Conclusion

Summary

- DataCollider detects and reports data races on x86
- Use of hardware breakpoints and sampling for low overhead
- Automatic pruning of most false positives
- Suitable for existing (kernel) code

Conclusion

Summary

- DataCollider detects and reports data races on x86
- Use of hardware breakpoints and sampling for low overhead
- Automatic pruning of most false positives
- Suitable for existing (kernel) code

Discussion

- Astonishingly simple approach
- Evaluation of overhead using a virtual machine!?
- `volatile` vs. synchronisation