"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism."
The OR’ing in of the CTXTF_NEED_CALLBACK flag can be swallowed by the AND’ing out of the CTXTF_RUNNING flag!
• Results in system hang.
Thread A

```assembly
mov eax, [pctxt->dwfCtxt]
and eax, NOT 10h
mov [pctxt->dwfCtxt], eax
EAX = ??
```

Thread B

```assembly
mov eax, [pctxt->dwfCtxt]
or eax, 20h
mov [pctxt->dwfCtxt], eax
EAX = ??
```

```
pctxt->dwfCtxt = 11h
```
Case study #1, assembled

Thread A

```
1 mov eax, [pctxt->dwarfCtx]
   and eax, NOT 10h
   mov [pctxt->dwarfCtx], eax

EAX = 11h
```

Thread B

```
mov eax, [pctxt->dwarfCtx]
   or eax, 20h
   mov [pctxt->dwarfCtx], eax

EAX = ??
```

pctxt->dwarfCtx = 11h
Case study #1, assembled

Thread A

1. `mov eax, [pctxt->dwfCtxt]`
2. `and eax, NOT 10h`
   `mov [pctxt->dwfCtxt], eax`
   EAX = 01h

Thread B

1. `mov eax, [pctxt->dwfCtxt]`
2. `or eax, 20h`
   `mov [pctxt->dwfCtxt], eax`
   EAX = ??

pctxt->dwfCtxt = 11h
Case study #1, assembled

Thread A

1. `mov eax, [pctxt->dwfCtxt]`
2. `and eax, NOT 10h`  
   /* CONTEXT SWITCH */
   `mov [pctxt->dwfCtxt], eax`

EAX = 01h

Thread B

`mov eax, [pctxt->dwfCtxt]`
`or eax, 20h`
`mov [pctxt->dwfCtxt], eax`

EAX = ??

pctxt->dwfCtxt = 11h
Case study #1, assembled

Thread A

1. \texttt{mov eax, [pctxt->dwarfCtx]}  
2. \texttt{and eax, NOT 10h}
3. /* CONTEXT SWITCH */
4. \texttt{mov [pctxt->dwarfCtx], eax}

EAX = 01h

Thread B

1. \texttt{mov eax, [pctxt->dwarfCtx]}  
2. \texttt{or eax, 20h}
3. \texttt{mov [pctxt->dwarfCtx], eax}

EAX = 11h

\texttt{pctxt->dwarfCtx} = 11h
Case study #1, assembled

Thread A

1. `mov eax, [pctxt->dwfCtxt]`
2. `and eax, NOT 10h
/* CONTEXT SWITCH */
3. `mov [pctxt->dwfCtxt], eax`

EAX = 01h

Thread B

3. `mov eax, [pctxt->dwfCtxt]`
4. `or eax, 20h`
5. `mov [pctxt->dwfCtxt], eax`

EAX = 31h

pctxt->dwfCtxt = 11h
Case study #1, assembled

Thread A

1. mov eax, [pctxt->dwfCtxt]
2. and eax, NOT 10h
   /* CONTEXT SWITCH */
3. mov [pctxt->dwfCtxt], eax
4. EAX = 01h

Thread B

3. mov eax, [pctxt->dwfCtxt]
4. or eax, 20h
5. mov [pctxt->dwfCtxt], eax
6. EAX = 31h

pctxt->dwfCtxt = 31h
Case study #1, assembled

Thread A

1. `mov eax, [pctxt->dwfCtxt]`
2. `and eax, NOT 10h /* CONTEXT SWITCH */`
3. `mov [pctxt->dwfCtxt], eax`

EAX = 01h

Thread B

3. `mov eax, [pctxt->dwfCtxt]`
4. `or eax, 20h`
5. `mov [pctxt->dwfCtxt], eax`

EAX = 31h

pctxt->dwfCtxt = 01h
Case study #1, assembled

Thread A

1. `mov eax, [pctxt->dwfCtxt]`
2. `and eax, NOT 10h`
3. `/* CONTEXT SWITCH */`
4. `mov [pctxt->dwfCtxt], eax`

Thread B

5. `mov eax, [pctxt->dwfCtxt]`
6. `or eax, 20h`
7. `mov [pctxt->dwfCtxt], eax`

EAX = 01h

CTXTF_NEED_CALLBACK disappeared!

(pctxt->dwfCtxt & 0x20 == 0)

EAX = 31h

pctxt->dwfCtxt = 01h
Windows case study #1

### Thread A

```
RunContext(...)
{
    pctxt->dwfCtxt &= ~CTXTF_RUNNING;
    and [ecx+40], ~10h
}
```

### Thread B

```
RestartCtxtCallback(...)
{
    pctxt->dwfCtxt |= CTXTF_NEED_CALLBACK;
    or [ecx+40], 20h
}
```

- Instructions appear atomic, but they are not!
* By our definition, a data race is a pair of memory accesses that satisfy all the below:

  * The accesses can happen concurrently

  * There is a non-zero overlap in the physical address ranges specified by the two accesses

  * At least one access modifies the contents of the memory location
Importance

* Very hard to reproduce
  * Timings can be very tight
* Hard to debug
  * Very easy to mistake as a hardware error “bit flip”

* To support scalability, code is moving away from monolithic locks
  * Fine-grained locks
  * Lock-free approaches
Previous Techniques

* Happens-before and lockset algorithms have significant overhead
  * Intel Thread Checker has 200x overhead
  * Log all synchronizations
  * Instrument all memory accesses

* High overhead can prevent usage in the field
  * Causes false failures due to timeouts
Prior schemes require a complete knowledge and logging of all locking semantics.

Locking semantics in kernel-mode can be homegrown, complicated and convoluted.
  * e.g. DPCs, interrupts, affinities
DataCollider: Goals
1. No false data races

- Tradeoff between having false positives and reporting fewer data races
False vs. Benign

* **False data race**
  * A data race that cannot actually occur

* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution
False vs. benign example

Thread A

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;

Thread B

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;
False vs. Benign

* **False data race**
  * A data race that cannot actually occur

* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution
False vs. benign example

Thread A

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;

Thread B

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;
2. User-controlled overhead

* Give user full control of overhead – from 0.0x up

* Fast vs. more races found
3. Actionable data

* Contextual information is key to analysis and debugging
Insights
1. **Instead of inferring** if a data race *could* have occurred, **let’s cause it to actually happen!**

* No locksets, no happens-before
2. **Sample memory accesses**
   * No binary instrumentation
     * No synchronization logging
     * No memory access logging
   * Use code and data breakpoints
   * Randomly selection for uniform coverage
Intersection Metaphor
Intersection Metaphor

Memory
Address = 0x1000
Intersection Metaphor

Hi, I'm Thread A!

Memory Address = 0x1000
Intersection Metaphor

Instruction stream

Memory Address = 0x1000
Intersection Metaphor

Instruction stream

I have the lock, so I get a green light.

Memory Address = 0x1000
Intersection Metaphor

Instruction stream

Memory
Address = 0x1000
Intersection Metaphor

Memory Address = 0x1000

DataCollider
Intersection Metaphor

DataCollider

Memory Address = 0x1000
Intersection Metaphor

Please wait a moment, Thread A – we’re doing a routine check for data races.

DataCollider
Intersection Metaphor

- Memory
  - Address = 0x1000
  - Value = 3

DataCollider

Data Breakpoint
Intersection Metaphor

Memory Address = 0x1000
Value = 3

DataCollider

Data Breakpoint
Intersection Metaphor: Normal Case
Intersection Metaphor: Normal Case

Memory Address = 0x1000
Value = 3

DataCollider

Data Breakpoint
Intersection Metaphor: Normal Case

Memory Address = 0x1000
Value = 3

Thread B

DataCollider

Data Breakpoint
Intersection Metaphor: Normal Case

I don’t’ have the lock, so I’ll have to wait.
Intersection Metaphor: Normal Case

Nothing to see here. Let me remove this trap.
Intersection Metaphor: Normal Case

Looks safe now. Sorry for the inconvenience.

DataCollider
Intersection Metaphor: Normal Case
Intersection Metaphor: Data Race
Intersection Metaphor: Data Race

DataCollider

Data Breakpoint
Intersection Metaphor: Data Race

Memory Address = 0x1000
Value = 3

Thread B

DataCollider

Data Breakpoint

DataCollider
Intersection Metaphor: Data Race

Memory Address = 0x1000 Value = 3

Locks are for wimps!

DataCollider

Data Breakpoint

Data Breakpoint
Intersection Metaphor: Data Race
Intersection Metaphor: Data Race
Intersection Metaphor: Data Race
Intersection Metaphor: Data Race

Looks safe now. Sorry for the inconvenience.
Intersection Metaphor: Data Race
Implementation
Sampling memory accesses with code breakpoints; part 1

### Process

1. Analyze target binary for memory access instructions.
2. Hook the breakpoint handler.
3. Set code breakpoints at a sampling of the memory access instructions.
4. Begin execution.

### Advantages

* Zero base-overhead—no code breakpoints means only the original code is running.
* No annotations required—only symbols.
OnCodeBreakpoint( pc ) {

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

    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 hit)
        ReportDataRace( );
}

Advantages

- Setting the data breakpoint will catch the colliding thread in the act.
- This provides much more actionable debugging information.
Advantages

- The additional re-read approach helps detect races caused by:
  - Hardware interaction via DMA
  - Physical memory that has multiple virtual mappings
Results
Most of dynamic data races are benign

Many have the potential to be heuristically pruned

Much room to investigate and develop in this area

<table>
<thead>
<tr>
<th>Data Race Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign – Heuristically Pruned</td>
<td></td>
</tr>
<tr>
<td>Statistic Counter</td>
<td>52</td>
</tr>
<tr>
<td>Safe Flag Update</td>
<td>29</td>
</tr>
<tr>
<td>Special Variable</td>
<td>5</td>
</tr>
<tr>
<td>Subtotal</td>
<td>86</td>
</tr>
<tr>
<td>Benign – Manually Pruned</td>
<td></td>
</tr>
<tr>
<td>Double-check locking</td>
<td>8</td>
</tr>
<tr>
<td>Volatile</td>
<td>8</td>
</tr>
<tr>
<td>Write Same Value</td>
<td>1</td>
</tr>
<tr>
<td>Other</td>
<td>1</td>
</tr>
<tr>
<td>Subtotal</td>
<td>18</td>
</tr>
<tr>
<td>Real</td>
<td></td>
</tr>
<tr>
<td>Confirmed Investigating</td>
<td>5</td>
</tr>
<tr>
<td>Subtotal</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>113</td>
</tr>
</tbody>
</table>
Results: bugs found

- 25 confirmed bugs in the Windows OS have been found

- 8 more are still pending investigation

<table>
<thead>
<tr>
<th>Data Races Reported</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>12</td>
</tr>
<tr>
<td>Confirmed and Being Fixed</td>
<td>13</td>
</tr>
<tr>
<td>Under Investigation</td>
<td>8</td>
</tr>
<tr>
<td>Harmless</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>38</td>
</tr>
</tbody>
</table>
Windows case study #2

Thread A

Connection->Initialized = TRUE;
or byte ptr [esi+70h],1

Thread B

Connection->QueuedForClosing = 1;
or byte ptr [esi+70h],2

struct CONNECTION {
    UCHAR Initialized : 1;
    UCHAR QueuedForClosing : 1;
};

This data race was found by using DataCollider on a test machine that was running a multi-threaded fuzzing test. It has been fixed.
Windows case study #3

Thread A (owns SpinLock)    Thread B

parentFdoExt->idleState = newState;  parentFdoExt->idleState = newState;

VOID ChangeIdleState(
    FDO_IDLE_STATE newState,
    BOOLEAN acquireLock);

This data race was found by using DataCollider on a test machine that was running a PnP stress test. In certain circumstances, ChangeIdleState was being called with acquireLock==FALSE even though the lock was not already acquired.
Results: Scalability

* By using the code breakpoint method, we can see that data races can be found with as little as 5% overhead
* The user can effectively adjust the balance between races found and overhead incurred
Future Work

* Better methods for prioritizing benign vs. non-benign races
  * Statistical analysis? Frequency?

* Apply algorithm to performance issues
  * True data sharing
  * False data sharing = data race “near miss”
DataCollider can detect data races

* with no false data races,

* with zero base-overhead,

* in kernel mode,

* and find real product bugs.

We’re hiring! 😊 jerick@microsoft.com
### DataCollider Original Prototype

**Original Algorithm**

```c
OnMemoryAccess( byte* Addr)
{
    if(rand() % 50 != 0) return;
    byte b = *Addr;
    int count = rand() % 1000;
    while(count--)
    {
        if(b != *Addr) Breakpoint();
    }
}
```

- "If the memory a thread is accessing changes, then a data race could have occurred."
- Used an internal tool to inject code into existing binaries
- Written without knowledge of lockset or happens-before approaches
False vs. benign example

Thread A:

```c
MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;
```

Thread B:

```c
MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;
```
**Issue:**

- Fixing a bug when one only has knowledge of one side of the race can be very time consuming because it would often require deep code review to find what the colliding culprit could be.

**Solution:**

- Make use of the hardware debug registers to cause a processor trap to occur on race.
Issue:

- Injecting code into a binary introduced an unavoidable non-trivial base overhead.

Solution:

- Dispose of injecting code into binaries entirely. Sample memory accesses via code breakpoints instead.
* **False data race**
  * A data race that is claimed to exist by a data race detection tool, but, in reality, cannot occur.

* **Benign data race**
  * A data race that can and does occur, but is intended to happen as part of normal program execution. E.g. synchronization primitives usually have benign data races as the key to their operation.

* **Real data race**
  * A data race that is not intended or causes unintended consequences. If the developer were to write the code again, he/she would do so differently.
False vs. benign vs. real example

Thread A

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;

Thread B

MyLockAcquire();
gReferenceCount++;
MyLockRelease();
gStatisticsCount++;
gReferenceCount++;