Towards Production-Run Heisenbugs Reproduction on Commercial Hardware

Shiyou Huang  Bowen Cai  and  Jeff Huang
What’s a coder’s worst nightmare?

https://www.quora.com/What-is-a-coders-worst-nightmare
The bug only occurs in production but cannot be replicated locally.

https://www.quora.com/What-is-a-coders-worst-nightmare
Heisenbug

When you trace them, they disappear!
Heisenbug

When you trace them, they disappear!

- Localization is hard
Heisenbug

When you trace them, they disappear!

- Localization is hard
- reproduction is hard
Heisenbug

When you trace them, they disappear!

- Localization is hard
- reproduction is hard
- never know if it is fixed...
A motivating example

Init: \(x=1, y=2\)

T1
1: T2.start()
2: \(z=0\)
3: \(x++\)
4: \(y++\)
5: \(z=1\)
6: T2.join()

T2
7: if \((z==1)\)
8: \assert(x+1==y)\)

\[x=2, y=3\]

contradiction!

http://stackoverflow.com/questions/16159203/
A motivating example

Init: $x=1$, $y=2$

T1
1: T2.start()
2: $z=0$
3: $x++$
4: $y++$
5: $z=1$
6: T2.join()

T2
7: if ($z==1$)
8: assert($x+1==y$)

contradiction!

PSO

HTTP://stackoverflow.com/questions/16159203/
A motivating example

$12$ million loss of equipment!

Init: $x=1$, $y=2$

1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

Contradiction!

x=2, y=3

x+1==y

http://stackoverflow.com/questions/16159203/
Record & Replay (RnR)

Goal: **record** the non-determinism at runtime and **reproduce** the failure.
Record & Replay (RnR)

Goal: *record* the non-determinism at runtime and *reproduce* the failure

- runtime overhead
- the ability to reproduce failures
Related Work

• Software-based approach
  • **order-based**: fully record shared memory dependencies at runtime
    • LEAP[FSE’10], Order[USENIX ATC’11], Chimera[PLDI’12], Light[PLDI’15] RR[USENIX ATC’17]...
    • Chimera: > 2.4x
  • **search-based**: partially record the dependencies at runtime and use offline analysis (e.g. SMT solvers) to reason the dependencies
    • ODR[SOSP’09], Lee et al. [MICRO’09], Weeratunge et al.[ASPLOS’10], CLAP[PLDI’13]...
    • CLAP: 0.9x – 3x

• Hardware-based approach
  • Rerun[ISCA’08], Delorean[ISCA’08], Coreracer[MICRO’11], PBI[ASPLOS’13]...
  • rely on special hardware that are not deployed
Reality of RnR

- high overheads
- failing to reproduce failures
- lack of commodity hardware support
Contributions

Goal: record the execution at runtime with low overhead and faithfully reproduce it offline

- RnR based on control flow tracing on commercial hardware (Intel PT)
- core-based constraints reduction to reduce the offline computation
- H3, evaluated on popular benchmarks and real-world applications, overhead: 1.4%-23.4%
Intel Processor Trace (PT)

**PT**: Program control flow tracing, supported on 5\textsuperscript{th} and 6\textsuperscript{th} generation Intel core

- Low overhead, as low as 5\%\textsuperscript{1}
- Highly compacted packets, <1 bit per retired instruction
- One bit (1/0) for branch taken indication
- Compressed branch target address

\textsuperscript{1}: https://sites.google.com/site/intelptmicrotutorial.
PT Tracing Overhead

<table>
<thead>
<tr>
<th>Program</th>
<th>Native time (s)</th>
<th>PT time (s)</th>
<th>PT OH(%)</th>
<th>trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>bodytrack</td>
<td>0.557</td>
<td>0.573</td>
<td>2.9%</td>
<td>94M</td>
</tr>
<tr>
<td>x264</td>
<td>1.086</td>
<td>1.145</td>
<td>5.4%</td>
<td>88M</td>
</tr>
<tr>
<td>vips</td>
<td>1.431</td>
<td>1.642</td>
<td>14.7%</td>
<td>98M</td>
</tr>
<tr>
<td>blackscholes</td>
<td>1.51</td>
<td>1.56</td>
<td>9.9%</td>
<td>289M</td>
</tr>
<tr>
<td>ferret</td>
<td>1.699</td>
<td>1.769</td>
<td>4.1%</td>
<td>145M</td>
</tr>
<tr>
<td>swaptions</td>
<td>2.81</td>
<td>2.98</td>
<td>6.0%</td>
<td>897M</td>
</tr>
<tr>
<td>raytrace</td>
<td>3.818</td>
<td>4.036</td>
<td>5.7%</td>
<td>102M</td>
</tr>
<tr>
<td>facesim</td>
<td>5.048</td>
<td>5.145</td>
<td>1.9%</td>
<td>110M</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>14.8</td>
<td>15.1</td>
<td>1.4%</td>
<td>1240M</td>
</tr>
<tr>
<td>freqmine</td>
<td>15.9</td>
<td>17.1</td>
<td>7.5%</td>
<td>2468M</td>
</tr>
<tr>
<td>Avg.</td>
<td>4.866</td>
<td>5.105</td>
<td>4.9%</td>
<td>553M</td>
</tr>
</tbody>
</table>

4.9% overhead on executions of PARSEC 3.0 on average
Challenges

• PT trace: low-level representation (assembly instruction)

• Absence of the thread information

• No data values of memory accesses
Solutions

• PT trace: low-level representation & no data values
  • Idea: extract the path profiles from PT trace and re-execute the program by KLEE to generate symbol values

• Absence of the thread information
  • Idea: use thread context switch information by Perf
H3 Overview

Phase 1: Control-flow tracing
Reconstruct the execution on each core by decoding the packets generated by PT and thread information from Perf

Phase 2: Offline analysis
- Path profiles of each thread
- Symbolic trace of each thread
- SMT constraints over the trace
Example

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
8: ✗ assert(x+1==y)

Step 1: Collecting path profiles of each thread

PT: tracing control-flow of the program’s execution

Binary image

libipt

Trace Packets + perf context switch events (TID, CPUID, TIME...)

T1

T2

line 1
line 2
... line n
Example

*Init*: $x = 1$, $y = 2$

1: T2.start()
2: $z = 0$
3: $x++$
4: $y++$
5: $z = 1$
6: T2.join()
7: if ($z == 1$)
8: $x++$
9: $y++$
10: $z = 1$
11: T2.join()
12: $x++$
13: $y++$
14: $z = 1$
15: T2.join()

*Step 1: Collecting path profiles of each thread*

*PT*: tracing control-flow of the program’s execution

**T1**: bb1

**T2**: bb1, bb2

Path profile:

- **T1**: bb1
- **T2**: bb1, bb2

Match to *.ll

path profile
Example

\textit{Init: } x=1, y=2

\begin{itemize}
  \item[T1]
    \begin{align*}
      &1: \text{T2.start()}
      &2: z=0
      &3: x++
      &4: y++
      &5: z=1
      &6: \text{T2.join()}
    \end{align*}

  \item[T2]
    \begin{align*}
      &7: \text{if (z==1)}
      &8: x \text{ assert(x+1==y)}
    \end{align*}
\end{itemize}

\textbf{Step 2: symbolic trace generation}

\textbf{KLEE[OSDI’08]: execute the thread along the path profile}

\begin{itemize}
  \item[T1]
    \begin{align*}
      W_z^2 &= 0 \\
      R_x^3, W_x^3 &= R_x^3 + 1 \\
      R_y^4, W_y^4 &= R_y^4 + 1 \\
      W_z^5 &= 1
    \end{align*}

  \item[T2]
    \begin{align*}
      \text{True} \equiv R_z^7 &= 1 \\
      R_x^8 + 1 \neq R_y^8
    \end{align*}
\end{itemize}

Using symbol values to represent concrete values, e.g.,

$W_z^2$: value written to $z$ at line 2

$R_x^3$: value read from $z$ at line 3
Example

Init: $x=1$, $y=2$

1: T2.start()
2: $z=0$
3: $x++$
4: $y++$
5: $z=1$
6: T2.join()
7: if ($z==1$)
8: $x$ assert($x+1==y$)

Step 3: computing global failure schedule

CLAP[PLDI’13]: Reason dependencies of memory accesses

Order variable $O$ represents the order of a statement, e.g., $O_2<O_3$ means $2:z=0$ happen before $3: x++$
Example

**Init:** \( x=1 \), \( y=2 \)

**T1**

1: T2.start()
2: \( z=0 \)
3: \( x++ \)
4: \( y++ \)
5: \( z=1 \)
6: T2.join()

**T2**

7: if \((z==1)\)
8: \( x \) assert\((x+1==y)\)

---

**Step 3: computing global failure schedule**

**CLAP[PLDI’13]:** Reason dependencies of memory accesses

Read-Write Constraints

\[
\begin{align*}
R_z^7 &= 0 \land O_7 < O_2 \lor \\
& \quad (R_z^7 = W_z^5 \land O_5 < O_7 \land (O_2 < O_5 \lor O_7 < O_2))
\end{align*}
\]

Memory Order Constraints

**SC**

\[
O_1 < O_2 < O_3^{R_X} < O_3^{W_X} < O_4^{R_X}
\]

\[
< O_4^{W_X} < O_5 < O_6
\]

\[
O_7 < O_8^x < O_8^y
\]

**PSO**

\[
O_1 < O_2 \quad O_5 < O_6\]

\[
O_3^{R_X} < O_3^{W_X} \quad O_4^{R_X} < O_4^{W_X}
\]

\[
O_7 < O_8^x < O_8^y
\]

Path Constraints

\[
R_z^7 = 1
\]

Failure Constraints

\[
R_x^8 + 1! = R_y^8
\]
Example

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
8: × assert(x+1==y)

Step 3: computing global failure schedule

CLAP[PLDI’13]: Reason dependencies of memory accesses

Read-Write Constraints

$R_z^7 = 0 \land Q_7 \leq Q_2 \lor$

$R_z^7 = W_z^5 \land Q_5 \leq Q_7 \land (Q_2 \leq Q_5 \lor Q_7 \leq Q_2)$

Memory Order Constraints

SC

$O_1 < O_2 < O_3^{Rx} < O_3^{Wx} < O_4^{Rx}$

$< O_4^{Wx} < O_5 < O_6$

$O_7 < O_8^x < O_8^y$

Path Constraints

$R_z^7 = 1$

PSO

$O_1 < O_2 < O_3^{Rx} < O_3^{Wx}$

$< O_4^{Rx} < O_6$

$O_7 < O_8^x < O_8^y$

Failure Constraints

$R_x^8 + 1! = R_y^8$

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
8: assert(x+1==y)
Example

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
8: assert(x+1==y)

Step 3: computing global failure schedule

CLAP[PLDI’13]: Reason dependencies of memory accesses

Read-Write Constraints

\[
(R^7_z = W_z^5 \land 0 < O_7 < 0_2 \land (O_2 < O_5 \lor O_7 < O_2))
\]

Memory Order Constraints

SC

\[
0_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x} < O_5 < O_6 < O_7 < O_8^y
\]

PSO

\[
O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x} < O_5 < O_6 < O_7 < O_8^y
\]

Path Constraints

\[
R^7_z = 1
\]

Failure Constraints

\[
R^8_x + 1! = R^8_y
\]
Example

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
8: x assert(x+1==y)

Step 3: computing global failure schedule

CLAP[PLDI’13]: Reason dependencies of memory accesses

Read-Write Constraints

$R_7^T = 0 \land O_7 < O_2 \lor (R_7^T = W_5^Z \land O_5 < O_7 \land (O_2 < O_5 \lor O_7 < O_2))$

Memory Order Constraints

SC

$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x} < O_5 < O_6$

Path Constraints

$R_7^T = 1$

PSO

$O_1 < O_2 < O_5 < O_6$

$O_3^{R_x} < O_3^{W_x} < O_4^{R_x} < O_4^{W_x}$

Failure Constraints

$R_8^y + 1 = R_8^y$

Execution should be allowed by the memory model.
Example

**Init**: \( x=1, \ y=2 \)

**T1**
1: \( T2.\text{start}() \)
2: \( z=0 \)
3: \( x++ \)
4: \( y++ \)
5: \( z=1 \)
6: \( T2.\text{join}() \)

**T2**
7: if \( (z==1) \)
8: \( x \) assert(\( x+1==y \))

---

**Step 3: computing global failure schedule**

**CLAP[PLDI'13]: Reason dependencies of memory accesses**

**Read-Write Constraints**

\[
(R^T_z = 0 \land O_7 < O_2) \lor \\
(R^T_z = W^5_z \land O_5 < O_7 \land (O_2 < O_5 \lor O_7 < O_2))
\]

**Memory Order Constraints**

**SC**

\[
0_1 < 0_2 < O^{Rx}_3 < O^{Wx}_3 < O^{Rx}_4 \\
< O^{Wx}_4 < O_5 < O_6 \\
O_7 < O^x_8 < O^y_8
\]

**PSO**

\[
0_1 < 0_2 \quad 0_5 < 0_6 \\
O^{Rx}_3 < O^{Wx}_3 \quad O^{Rx}_4 < O^{Wx}_4 \\
O_7 < O^x_8 < O^y_8
\]

**Path Constraints**

\[
R^7_z = 1
\]

**Failure Constraints**

\[
R^8_x + 1! = R^8_y
\]

*make the failure happen*
Example

Init: \( x=1, \ y=2 \)

1: \( T_2.\text{start}() \)
2: \( z=0 \)
3: \( x++ \)
4: \( y++ \)
5: \( z=1 \)
6: \( T_2.\text{join}() \)
7: if \( z==1 \)
8: \( \times \) assert\((x+1==y)\) violate

Step 3: computing global failure schedule

CLAP[PLDI’13]: Reason dependencies of memory accesses

Read-Write Constraints

\[ (R^7_Z = 0 \land O_7 < O_2) \lor (R^7_Z = W^5_Z \land O_5 < O_7 \land (O_2 < O_5 \lor O_7 < O_2)) \]

Memory Order Constraints

SC
\[
\begin{align*}
0_1 &< 0_2 < O^R_3 < O^W_3 < O^R_4 \\
&< O^W_4 < O_5 < O_6 \\
O_7 &< O^x_8 < O^y_8
\end{align*}
\]

PSO
\[
\begin{align*}
0_1 &< O_2 < 0_5 < 0_6 \\
O^R_3 &< O^W_3 < O^R_4 < O^W_4 \\
O_7 &< O^x_8 < 0^y_8
\end{align*}
\]

Path Constraints
\( R^7_Z = 1 \)

Failure Constraints
\( R^8_x + 1! = R^8_y \)

Violation

make the failure happen
Example

Init: x=1, y=2

T1
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2
7: if (z==1)
   x assert(x+1==y)

Step 3: computing global failure schedule

O_1=1, O_2=2, O_3=3, O_5=4, O_7=5, O_8=6, O_4=7

Schedule:
1-2-3-5-7-8-4

HEISENBUG! SOLVED!
TIME FOR A BEER
Core-based constraints reduction

- All the writes write a different value to the same memory location

Match R to the write $W_7$
Core-based constraints reduction

Without the partial order on each core

$W_1$, $W_2$, $W_3$, ..., $W_{15}$, $W_{16}$
Core-based constraints reduction

Without the partial order on each core
Core-based constraints reduction

Without the partial order on each core
Core-based constraints reduction

Without the partial order on each core

\[ W_1 \quad W_2 \quad W_3 \quad \ldots \quad W_{15} \quad W_{16} \]

\[ W_7-R \]
Core-based constraints reduction

Without the partial order on each core

![Diagram showing relationships between W1, W2, W3, ..., W15, W16, and W7-R with 2^{15} connections.](image-url)
Core-based constraints reduction

Knowing the partial order on each core

W_7-R

W_1  W_2  W_3  W_4

...  ...  

W_{13}  W_{14}  W_{15}  W_{16}
Core-based constraints reduction

Knowing the partial order on each core

\[ W_1 \quad W_2 \quad W_3 \quad W_4 \]

...  

...  

\[ W_{13} \quad W_{14} \quad W_{15} \quad W_{16} \]
Core-based constraints reduction

Knowing the partial order on each core

\[ W_{7-R} \]

\[ W_1 \quad W_2 \quad W_3 \quad W_4 \]

\[ \ldots \]

\[ W_{13} \quad W_{14} \quad W_{15} \quad W_{16} \]

\[ 5 \]

\[ 5^4 \quad \text{reduced from } 2^{15} \]
H3 Implementation

• Control-flow tracing
  • PT decoding library & Linux Perf tool

• Path profiles generation
  • Python scripts to extract the path profiles from PT trace

• Symbolic trace collecting
  • Modified KLEE[OSDI’08] for symbolic execution along the path profiles

• Constraints construction
  • Modified CLAP[PLDI’13] to implement the core-based constraints reduction
  • Z3 for solving the constraints
Evaluation

• Environment
  • 4 core 3.5GHz Intel i7 6700HQ Skylake with 16 GB RAM
  • Ubuntu 14.04, Linux kernel 4.7

• Three sets of experiments
  • runtime overhead
  • how effective to reproduce bugs
  • how effective is the core-based constraints reduction
### Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>#Threads</th>
<th>#SV</th>
<th>#insns (executed)</th>
<th>#branches (total)</th>
<th>#branches (app)</th>
<th>Ratio app/total</th>
<th>Symb. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>racey</td>
<td>192</td>
<td>4</td>
<td>3</td>
<td>1,229,632</td>
<td>78,117</td>
<td>77,994</td>
<td>99.8%</td>
<td>107s</td>
</tr>
<tr>
<td>pfscan</td>
<td>1026</td>
<td>3</td>
<td>13</td>
<td>1,287</td>
<td>237</td>
<td>43</td>
<td>18.1%</td>
<td>2.5s</td>
</tr>
<tr>
<td>aget-0.4.1</td>
<td>942</td>
<td>4</td>
<td>30</td>
<td>3,748</td>
<td>313</td>
<td>5</td>
<td>1.6%</td>
<td>117s</td>
</tr>
<tr>
<td>pbzip2-0.9.4</td>
<td>1942</td>
<td>5</td>
<td>18</td>
<td>1,844,445</td>
<td>272,453</td>
<td>5</td>
<td>0.0018%</td>
<td>8.7s</td>
</tr>
<tr>
<td>bbuf</td>
<td>371</td>
<td>5</td>
<td>11</td>
<td>1,235</td>
<td>257</td>
<td>3</td>
<td>1.2%</td>
<td>5.5s</td>
</tr>
<tr>
<td>sbuf</td>
<td>151</td>
<td>2</td>
<td>5</td>
<td>64,993</td>
<td>11,170</td>
<td>290</td>
<td>2.6%</td>
<td>1.6s</td>
</tr>
<tr>
<td>httpd-2.2.9</td>
<td>643K</td>
<td>10</td>
<td>22</td>
<td>366,665</td>
<td>63,653</td>
<td>12,916</td>
<td>20.3%</td>
<td>712s</td>
</tr>
<tr>
<td>httpd-2.0.48</td>
<td>643K</td>
<td>10</td>
<td>22</td>
<td>366,379</td>
<td>63,809</td>
<td>13,074</td>
<td>20.5%</td>
<td>698s</td>
</tr>
<tr>
<td>httpd-2.0.46</td>
<td>643K</td>
<td>10</td>
<td>22</td>
<td>366,271</td>
<td>63,794</td>
<td>12,874</td>
<td>20.2%</td>
<td>643s</td>
</tr>
</tbody>
</table>

http://pages.cs.wisc.edu/~markhill/racey.html
https://github.com/jiedy/concurrency-bugs
Runtime overhead

Comparison between H3 and CLAP

Runtime overhead

- Racey: H3 7.50%, CLAP 186.60%
- Pfscan: H3 11%, CLAP 23.40%
- Aget: H3 12.10%, CLAP 9.40%
- Pbizp2: H3 31.40%, CLAP 12.90%
- Bbuf: H3 9.80%, CLAP 38.50%
- Sbuf: H3 20%, CLAP 18.50%
- Httpd1: H3 13.80%, CLAP 34%
- Httpd2: H3 7.50%, CLAP 32.10%
- Httpd3: H3 12.90%, CLAP 13.30%

CLAP and H3 runtime overhead comparison.
Runtime overhead

Comparison between H3 and CLAP

CLAP: 64.3% vs H3: 12.9%
reduction: 31.3%
Constraints reduction

Core-based constraints reduction by H3 to CLAP

Reduced by > 30%

Reduced by > 90%

#Constraints

bbuf  sbuf  pfsan  pbzip2  racey1  racey2  racey3

CLAP  H3
Bug reproduction

Core-based constraints reduction by H3 to CLAP

- Reproduced by both
- Only reproduced by H3

#Constraints
Conclusion

H3: Reproducing Heisenbugs based on control flow tracing on commercial hardware (Intel PT)

• Runtime Overhead
  • PARSEC 3.0: ~4.9%
  • Real application: ~12.9% vs CLAP[PLDI’13] ~64.3%

• Bug reproduction
  • reproduces one more bug than CLAP
Discussion

• Symbolic execution is slow
  • Eliminate symbolic execution: use hardware watchpoints to catch values and memory locations

• Constraints for long traces
  • Use checkpoints and periodic global synchronization

• Non-deterministic program inputs (e.g., syscall results)
  • Integrate with Mozilla RR [USENIX ATC’17]
  • Key insight: use H3 to handle schedules, and RR to handle inputs
Thank you