Towards Production-Run Heisenbugs Reproduction on Commercial Hardware

Shiyou Huang Bowen Cai and Jeff Huang

Parasol ASER
Automated Software Engineering Research

TEXAS A&M UNIVERSITY

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!

NOT SURE IF I FIXED HEISENBUG
OR IF ITS JUST NOT HAPPENING NOW THAT I AM TRYING TO FIX IT
quickmeme.com

Heisenbug

When you trace them, they disappear!

- Localization is hard

NOT SURE IF I FIXED HEISENBUG
OR IF ITS JUST NOT HAPPENING NOW THAT I AM TRYING TO FIX IT
quickmeme.com

Heisenbug

When you trace them, they disappear!

- Localization is hard
- reproduction is hard

NOT SURE IF I FIXED HEISENBUG
OR IF ITS JUST NOT HAPPENING NOW THAT I AM TRYING TO FIX IT
quickmeme.com

Heisenbug

When you trace them, they disappear!

- Localization is hard
- reproduction is hard
- never know if it is fixed…

NOT SURE IF I FIXED HEISENBUG
OR IF ITS JUST NOT HAPPENING NOW THAT I AM TRYING TO FIX IT
quickmeme.com

A motivating example

Init: x=1, y=2

```
     T1                     T2
1: T2.start()         7: if (z==1)
                                       ✗
2: z=0                8:     assert(x+1==y)
3: x++
4: y++
5: z=1
6: T2.join()
```

z=1

contradiction!

x=2, y=3

x+1==y

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()
```

PSO

T2

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

✗

z=1

contradiction!

x=2, y=3

x+1==y

http://stackoverflow.com/questions/16159203/

A motivating example

Init: x=1, y=2

T1

1: T2.sta

2: z=0

3: x++

4: y++

5: z=1

6: T2.join()

$12 million loss of equipment!

contradiction!

x=2, y=3

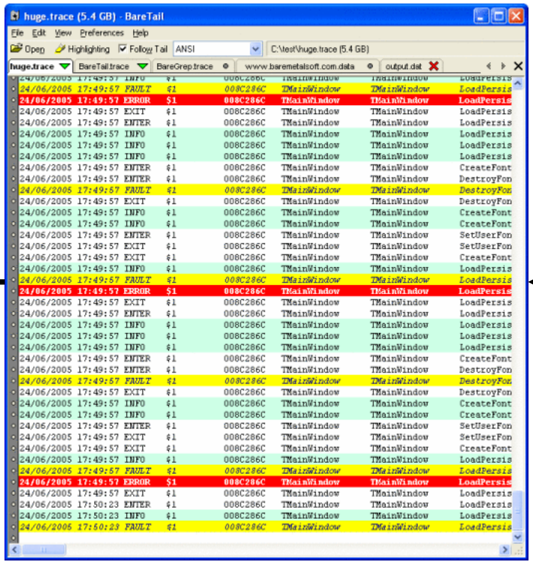
x+1==y

http://stackoverflow.com/questions/16159203/

Record & Replay (RnR)

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

Replay


huge.trace (5.4 GB) - BareTail
File Edit View Preferences Help
Open Highlighting Follow Tail ANSI
C:\test\huge.trace (5.4 GB)
huge.trace BareTail.trace BareGrep.trace www.baremetalsoft.com.data output.dat

```
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 FAULT  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ERROR  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 FAULT  $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  SetUserFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  SetUserFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 FAULT  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ERROR  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 FAULT  $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  DestroyFon
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 ENTER  $1   008C286C  TMainWindow  TMainWindow  SetUserFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  SetUserFon
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  CreateFont
24/06/2005 17:49:57 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 FAULT  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 ERROR  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:49:57 EXIT   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:50:23 ENTER  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:50:23 INFO   $1   008C286C  TMainWindow  TMainWindow  LoadPersis
24/06/2005 17:50:23 FAULT  $1   008C286C  TMainWindow  TMainWindow  LoadPersis
```

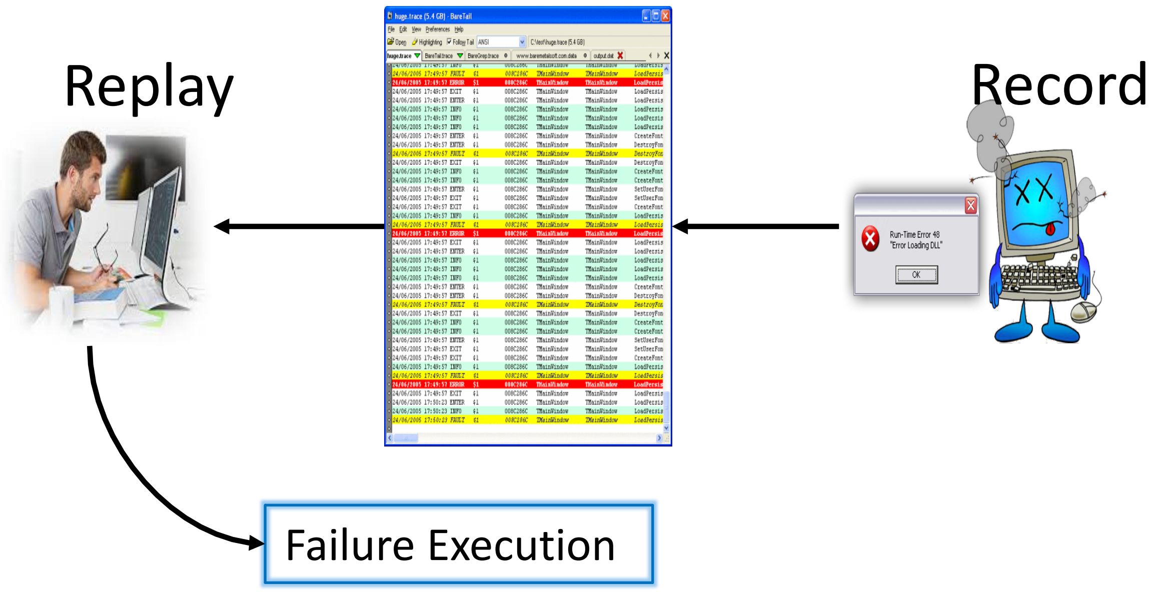
Record

Run-Time Error 48
"Error Loading DLL"

Failure Execution

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

In production

- 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 5th and 6th generation *Intel* core

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

1: https://sites.google.com/site/ intelptmicrotutorial.

PT Tracing Overhead

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

4.9% overhead on executions of PARSEC 3.0 on average

Challenges

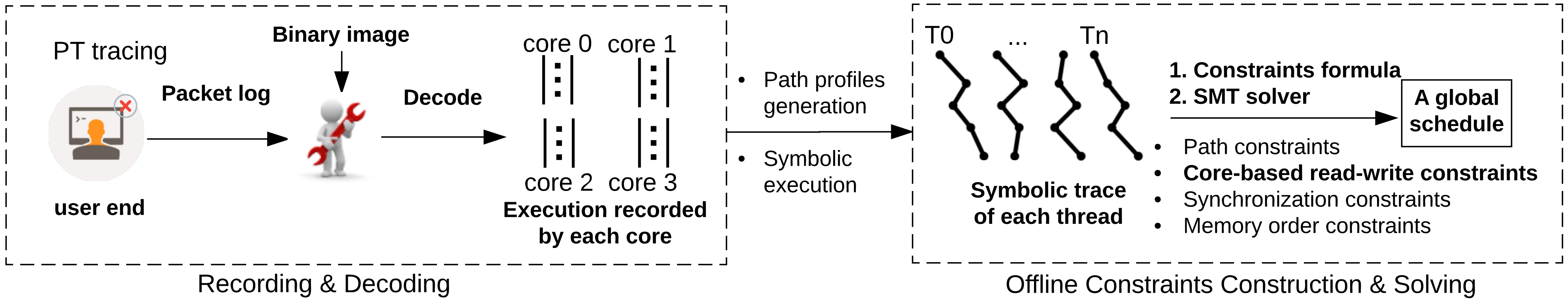
- PT trace: low-level representation (assembly instruction)
- Absence of the thread information
- No data values of memory accesses

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

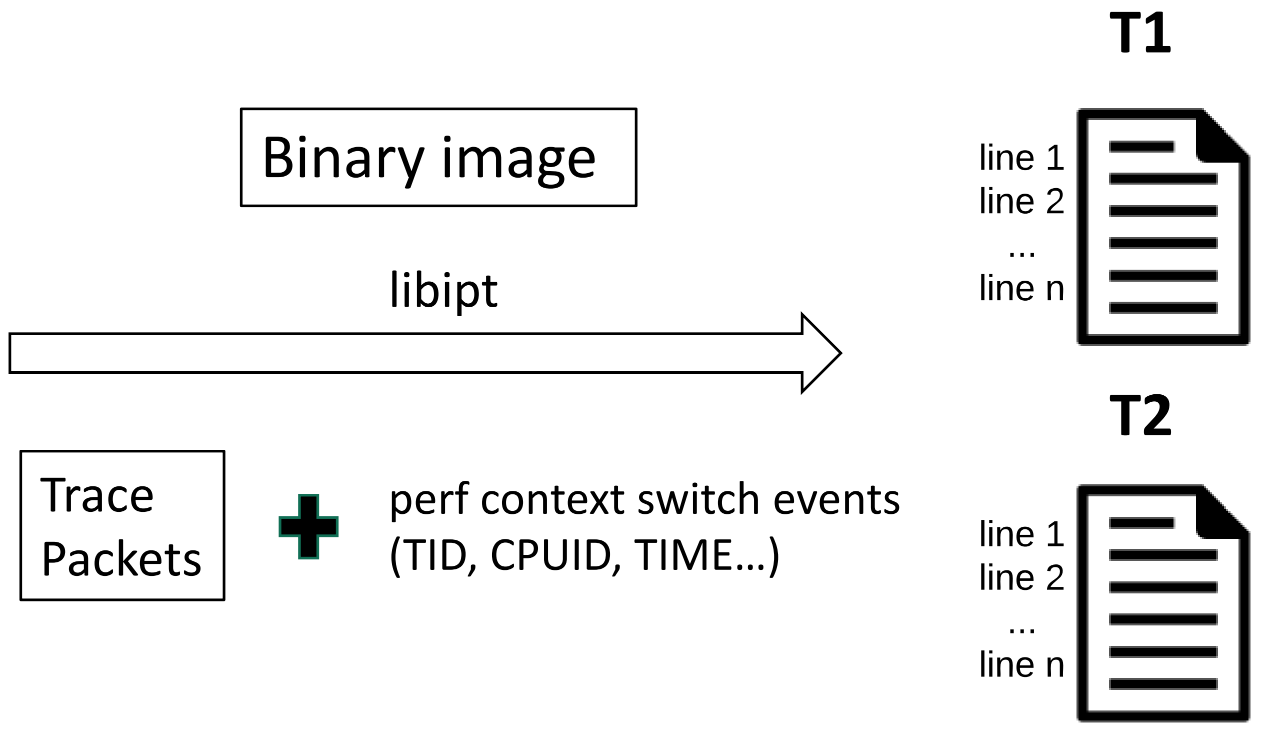
- Path profiles of each thread
- Symbolic trace of each thread
- SMT constraints over the trace

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)
```

Step1: Collecting path profiles of each thread

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



Example

Init: x=1, y=2

T1

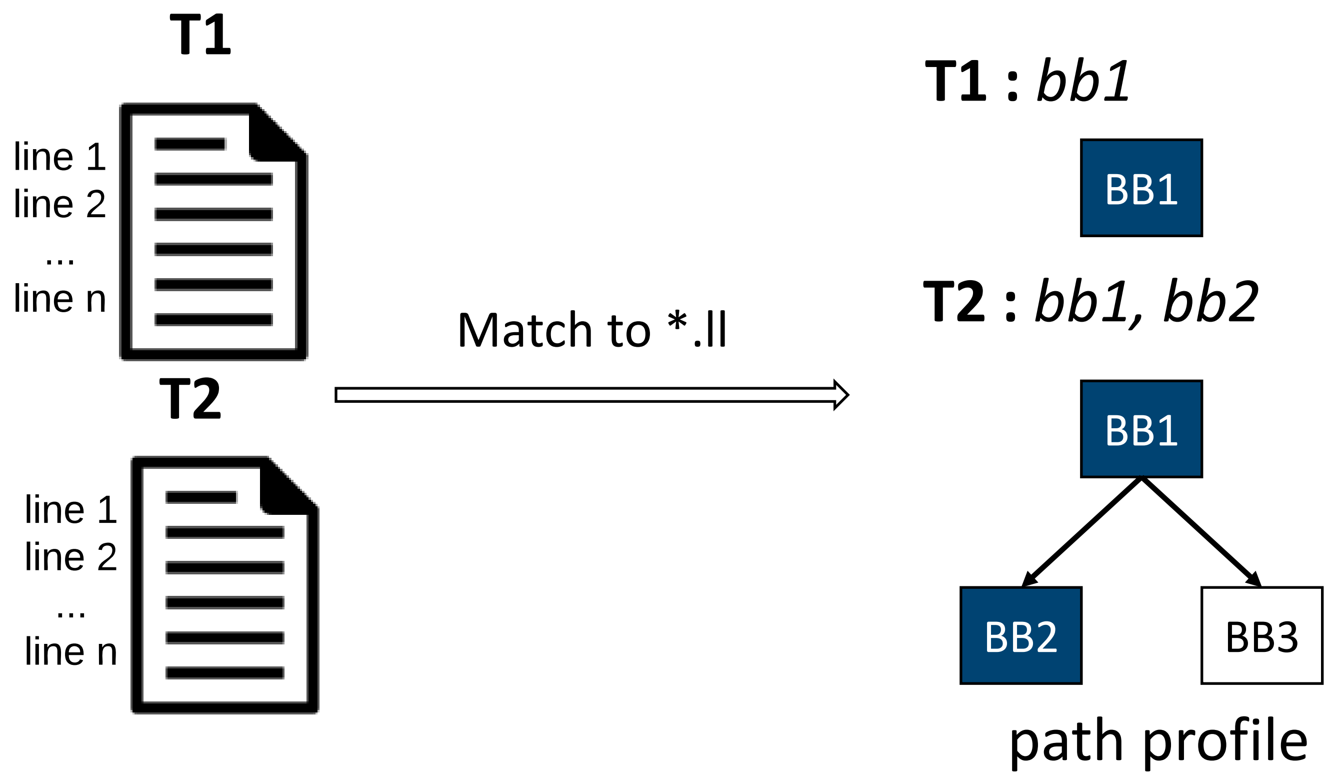
1: T2.start()
2: z=0
3: x++
4: y++
5: z=1
6: T2.join()

T2

7: if (z==1)
8: ✗ assert(x+1==y)

Step1: Collecting path profiles of each thread

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



Example

Step2: symbolic trace generation

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)
```

KLEE[OSDI'08]: execute the thread along the path profile

T1

$W_z^2 = 0$

$R_x^3, W_x^3 = R_x^3 + 1$

$R_y^4, W_y^4 = R_y^4 + 1$

$W_z^5 = 1$

T2

$True \equiv R_z^7 == 1$

$R_x^8 + 1 \neq R_y^8$

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

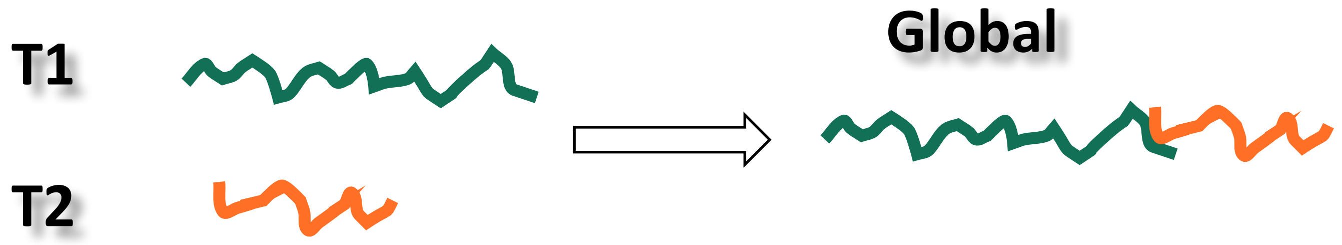
Example

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



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:    ✗ assert(x+1==y)
```

Step 3: computing global failure schedule

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

Read-Write Constraints

match a read to a write

$$(R_z^7 = 0 \ \wedge O_7 < O_2) \vee$$
$$(R_z^7 = W_z^5 \ \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{Rx} < O_3^{Wx} < O_4^{Rx}$$
$$< O_4^{Wx} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

PSO

$$O_1 < O_2 \quad O_5 < O_6$$
$$O_3^{Rx} < O_3^{Wx} \quad O_4^{Rx} < O_4^{Wx}$$
$$O_7 < O_8^x < O_8^y$$

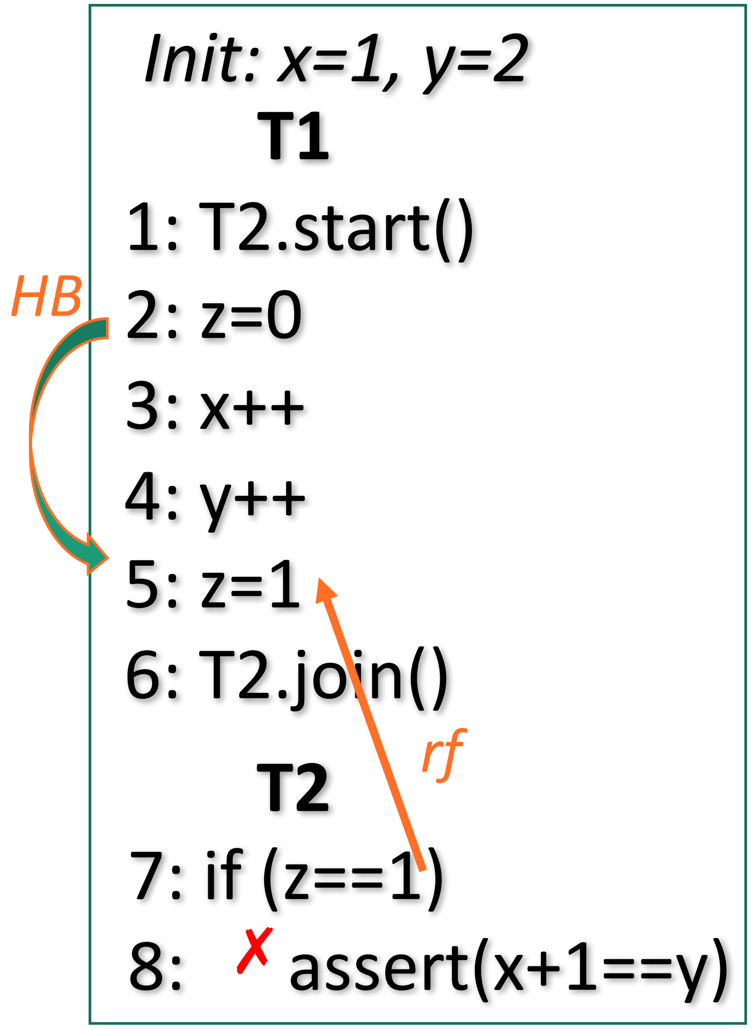
Path Constraints

$$R_z^7 = 1$$

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

match a read to a write

$$(R_z^7 = 0 \ \land O_7 < O_2) \lor$$
$$\boxed{(R_z^7 = W_z^5 \ \land O_5 < O_7 \land (O_2 < O_5 \lor O_7 < O_2))}$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x}$$
$$< O_4^{W_x} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

PSO

$$O_1 < O_2 \quad O_5 < O_6$$
$$O_3^{R_x} < O_3^{W_x} \quad O_4^{R_x} < O_4^{W_x}$$
$$O_7 < O_8^x < O_8^y$$

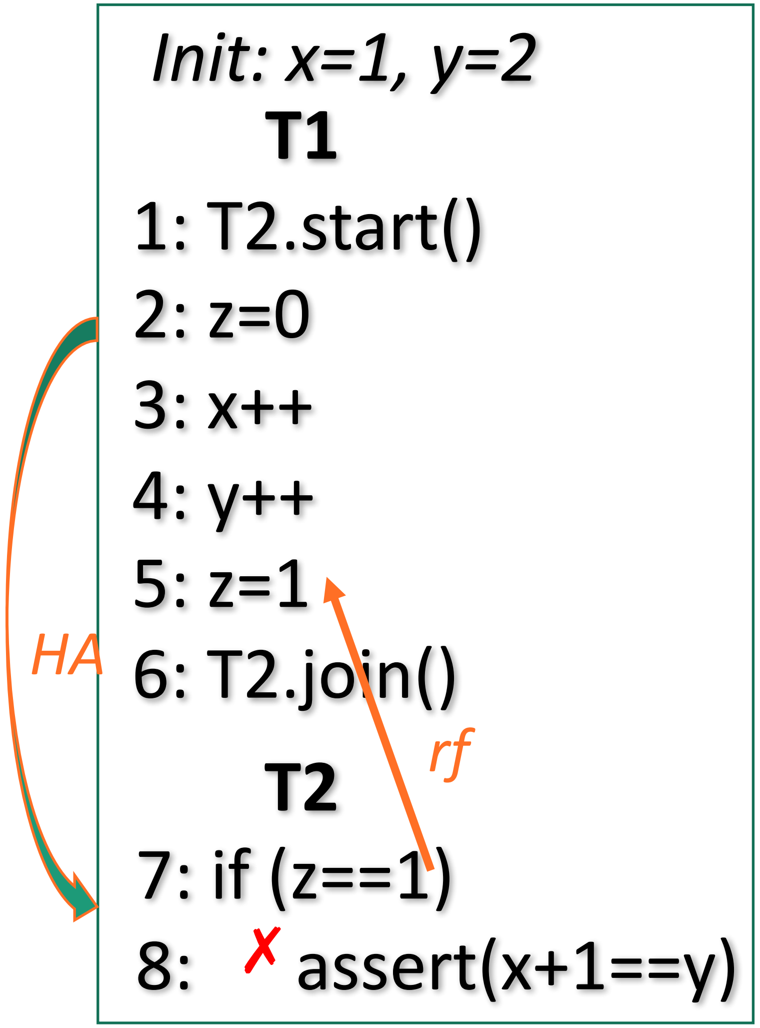
Path Constraints

$$R_z^7 = 1$$

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

$$(R_z^7 = 0 \wedge O_7 < O_2) \vee$$
$$\boxed{(R_z^7 = W_z^5 \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))}$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x}$$
$$< O_4^{W_x} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

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()
```

reordering PSO

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

$$(R_z^7 = 0 \wedge O_7 < O_2) \vee$$
$$(R_z^7 = W_z^5 \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{R_x} < O_3^{W_x} < O_4^{R_x}$$
$$< O_4^{W_x} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

PSO

$$O_1 < O_2 \quad O_5 < O_6$$
$$O_3^{R_x} < O_3^{W_x} \quad O_4^{R_x} < O_4^{W_x}$$
$$O_7 < O_8^x < O_8^y$$

execution should be allowed by the memory model

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        True
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

$$(R_z^7 = 0 \ \wedge O_7 < O_2) \vee$$
$$(R_z^7 = W_z^5 \ \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{Rx} < O_3^{Wx} < O_4^{Rx}$$
$$< O_4^{Wx} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

PSO

$$O_1 < O_2 \quad O_5 < O_6$$
$$O_3^{Rx} < O_3^{Wx} \quad O_4^{Rx} < O_4^{Wx}$$
$$O_7 < O_8^x < O_8^y$$

Path Constraints

$$R_z^7 = 1$$

Failure Constraints

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

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)
8:   ✗ assert(x+1==y)
```

Violation

Step 3: computing global failure schedule

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

Read-Write Constraints

$$(R_z^7 = 0 \ \wedge O_7 < O_2) \vee$$
$$(R_z^7 = W_z^5 \ \wedge O_5 < O_7 \wedge (O_2 < O_5 \vee O_7 < O_2))$$

Memory Order Constraints

SC

$$O_1 < O_2 < O_3^{Rx} < O_3^{Wx} < O_4^{Rx}$$
$$< O_4^{Wx} < O_5 < O_6$$
$$O_7 < O_8^x < O_8^y$$

PSO

$$O_1 < O_2 \quad O_5 < O_6$$
$$O_3^{Rx} < O_3^{Wx} \quad O_4^{Rx} < O_4^{Wx}$$
$$O_7 < O_8^x < O_8^y$$

Path Constraints

$$R_z^7 = 1$$

Failure Constraints

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

make the failure happen

Example

Init: x=1, y=2

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

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

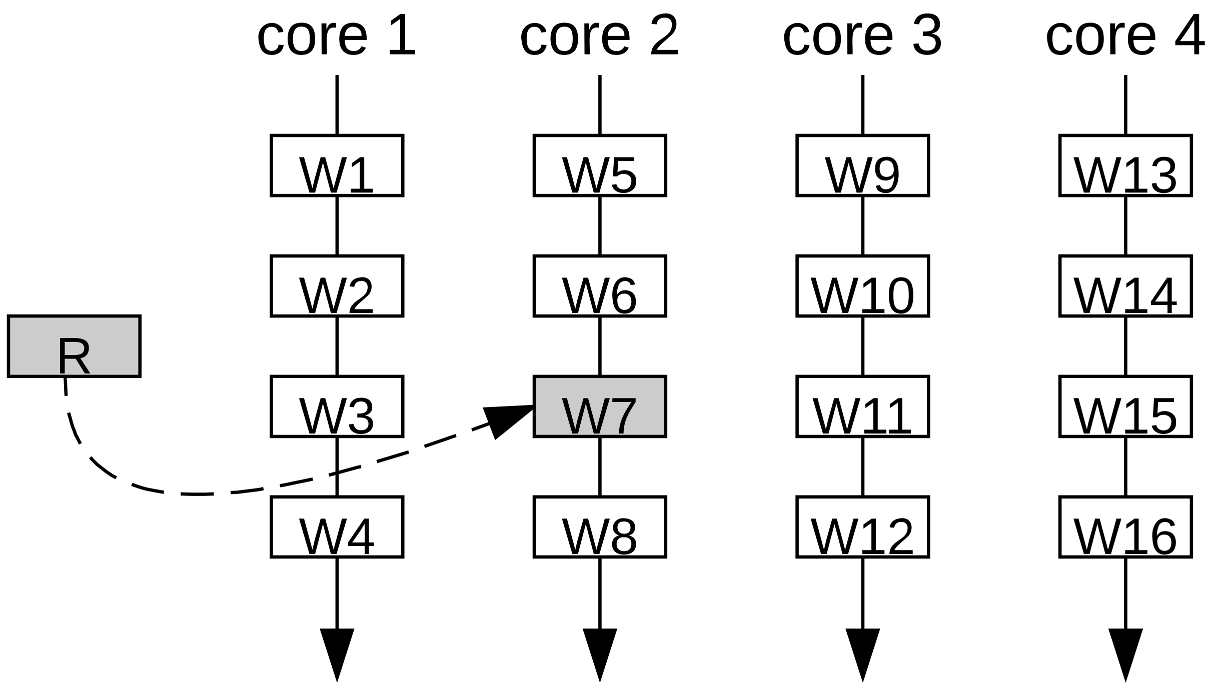
Step 3: computing global failure schedule

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

Schedule:

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

Core-based constraints reduction

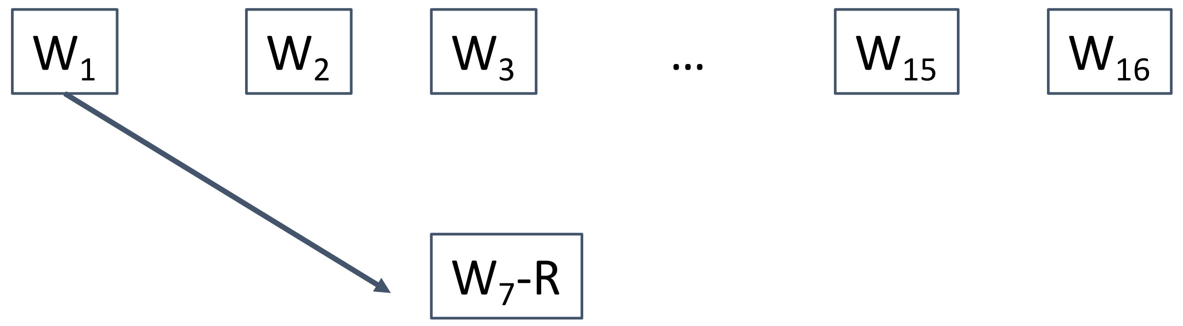


Match R to the write W_7

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

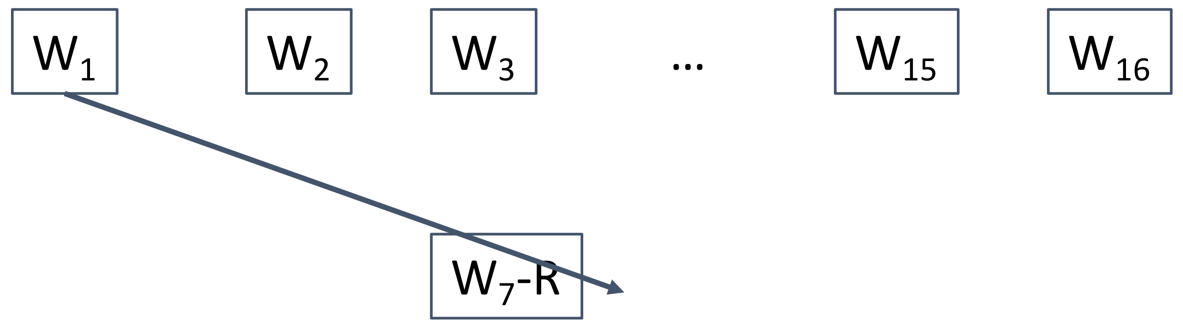
Core-based constraints reduction

Without the partial order on each core



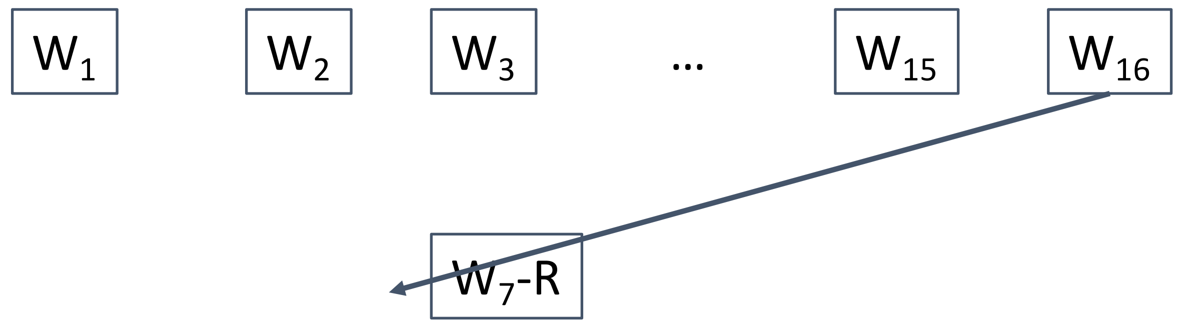
Core-based constraints reduction

Without the partial order on each core



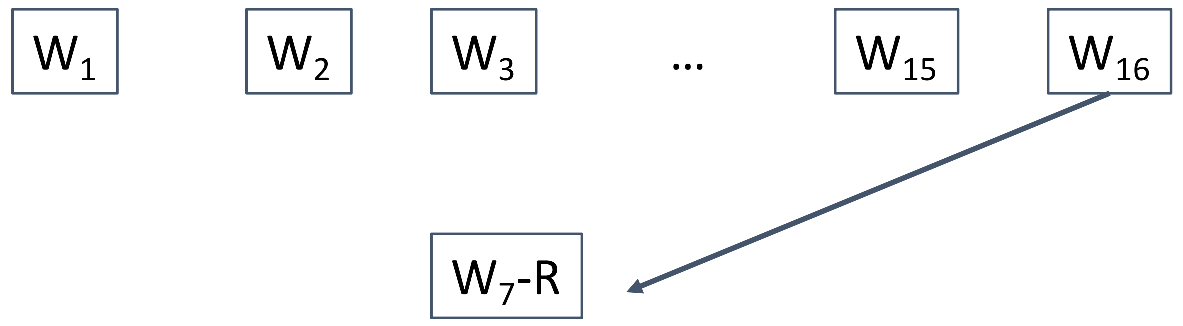
Core-based constraints reduction

Without the partial order on each core



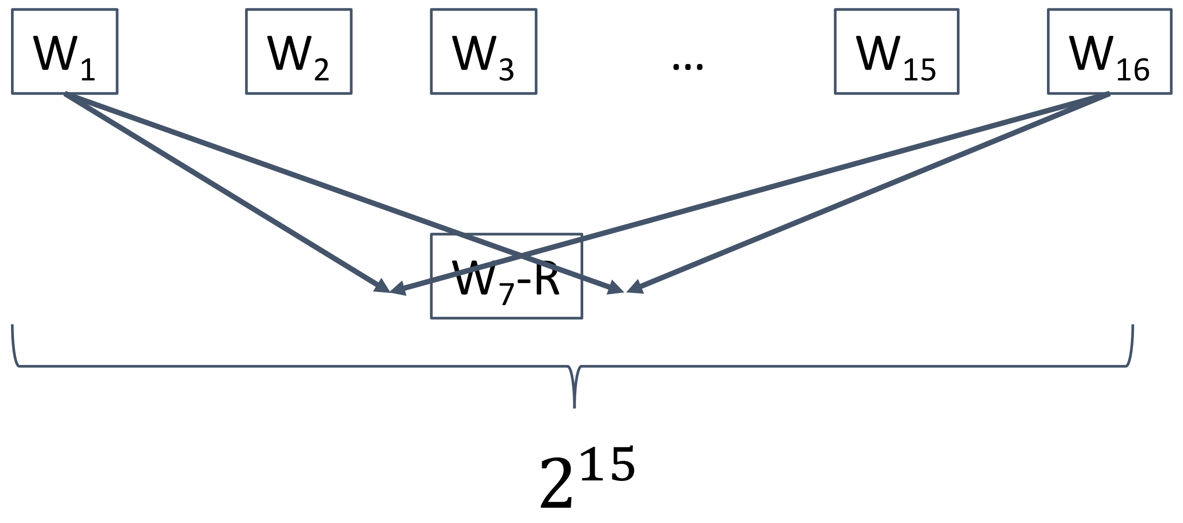
Core-based constraints reduction

Without the partial order on each core



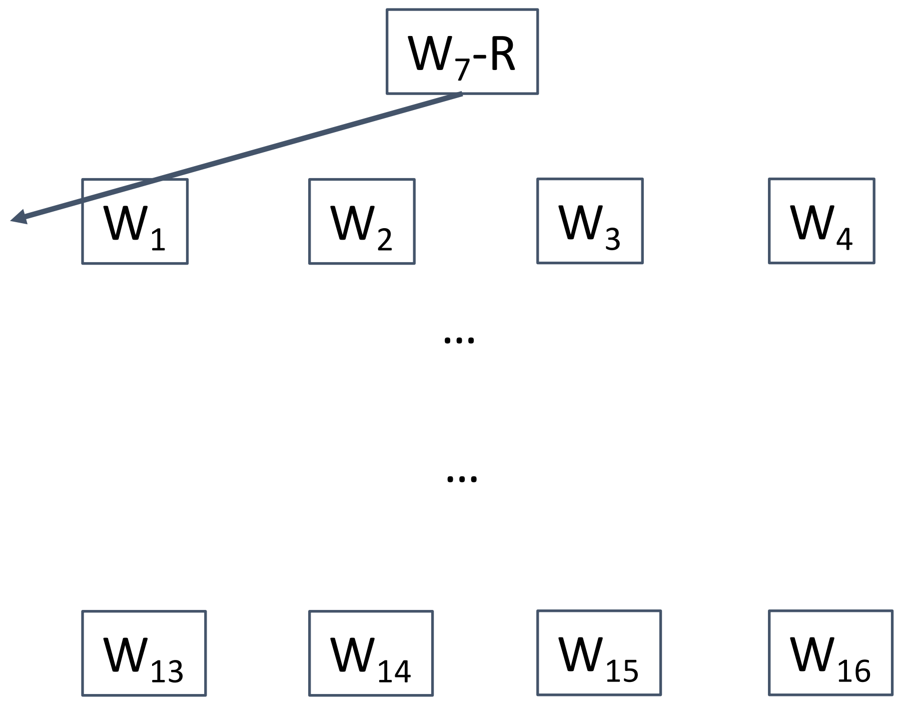
Core-based constraints reduction

Without the partial order on each core



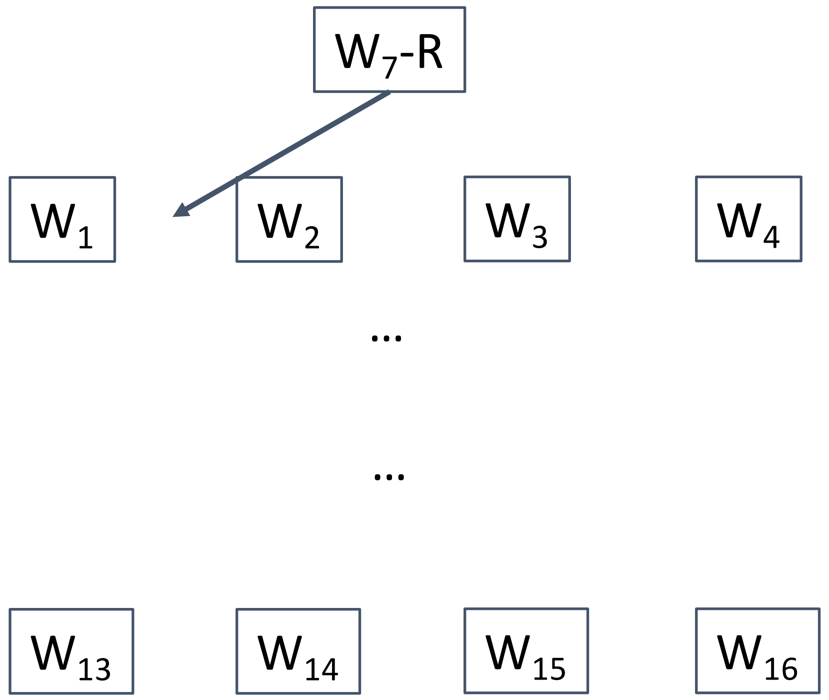
Core-based constraints reduction

Knowing the partial order on each core



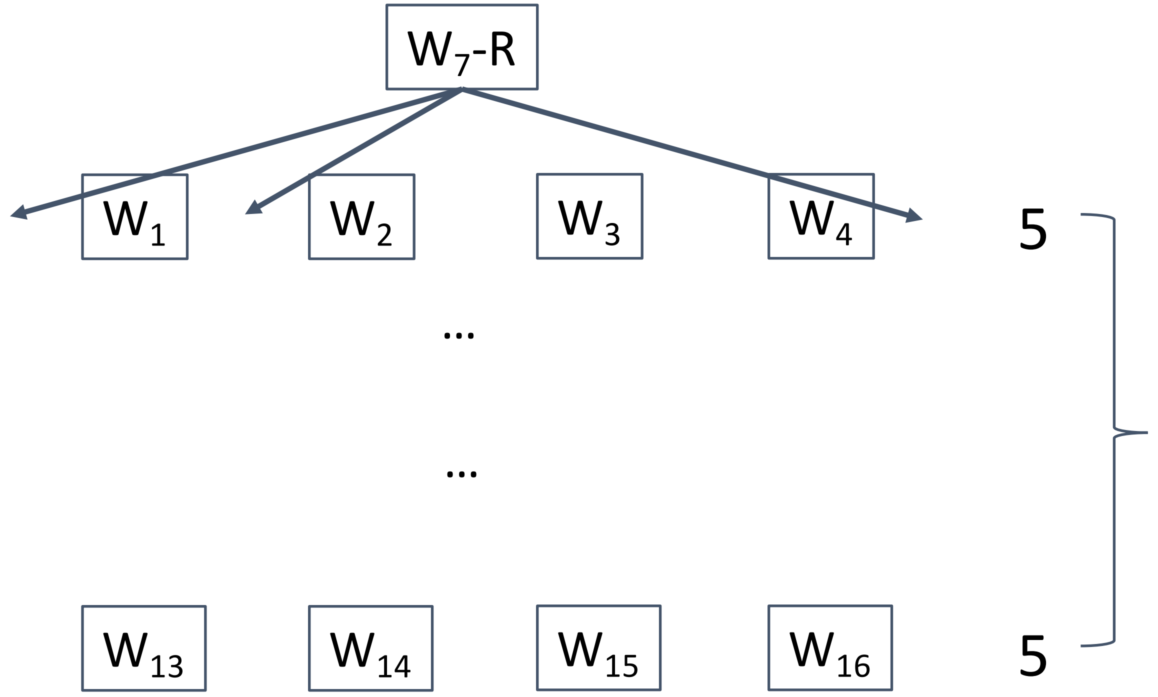
Core-based constraints reduction

Knowing the partial order on each core



Core-based constraints reduction

Knowing the partial order on each core



5^4 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

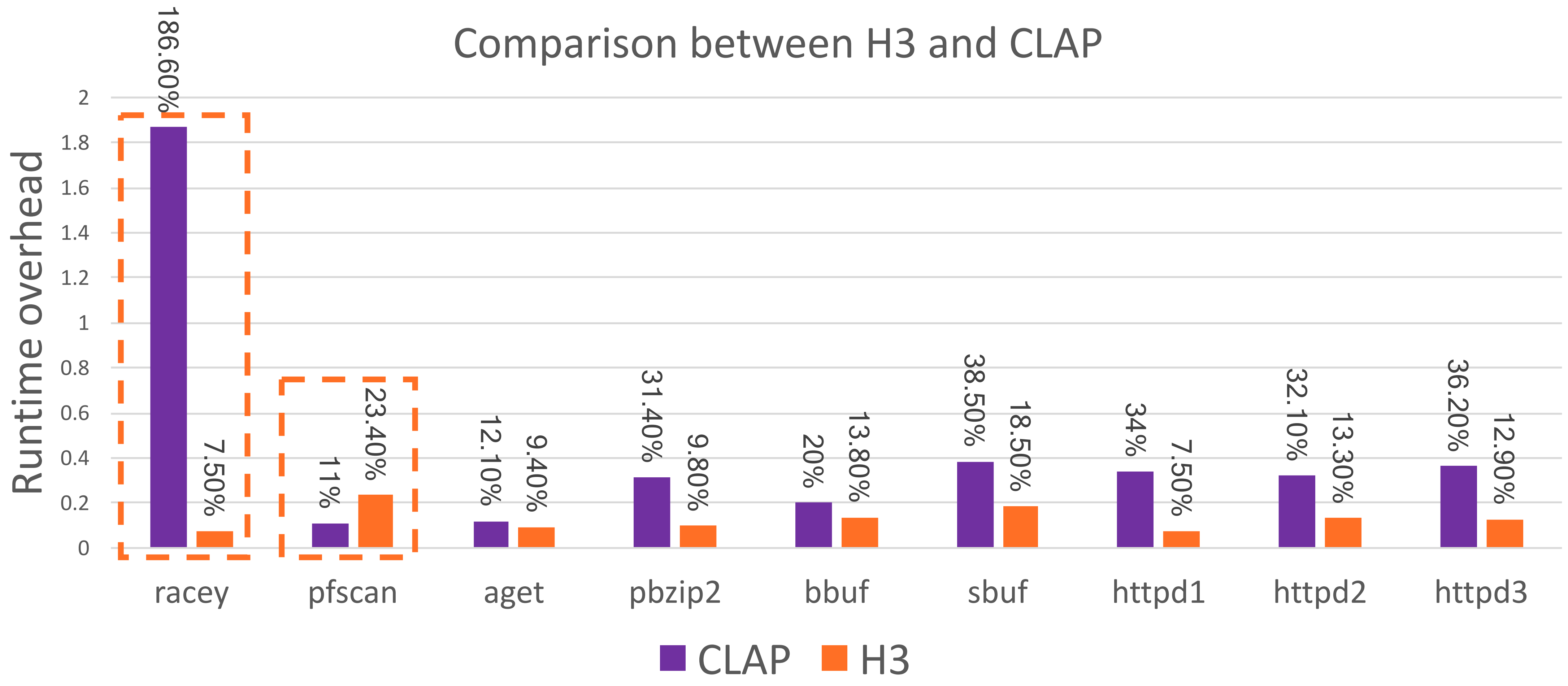
Benchmarks

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

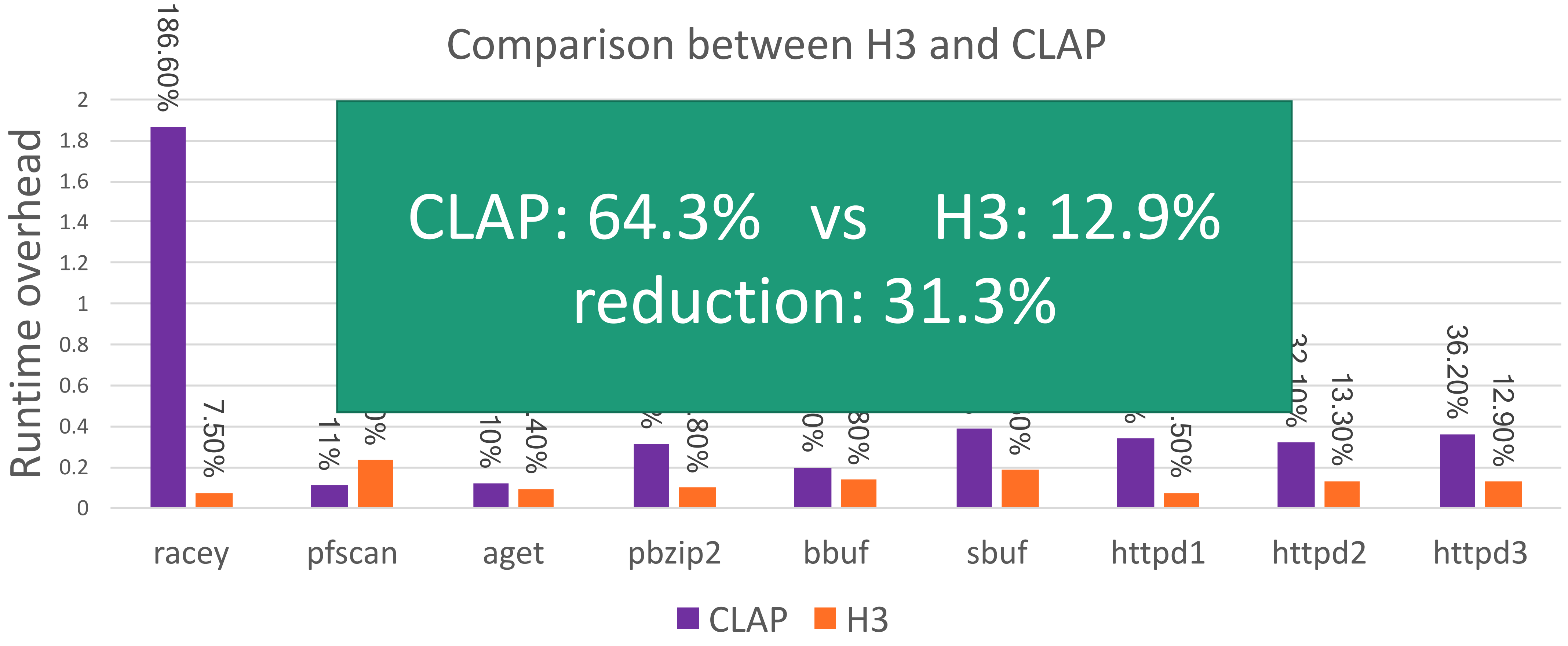
http://pages.cs.wisc.edu/~markhill/ racey.html

https://github.com/jieyu/concurrency-bugs

Runtime overhead

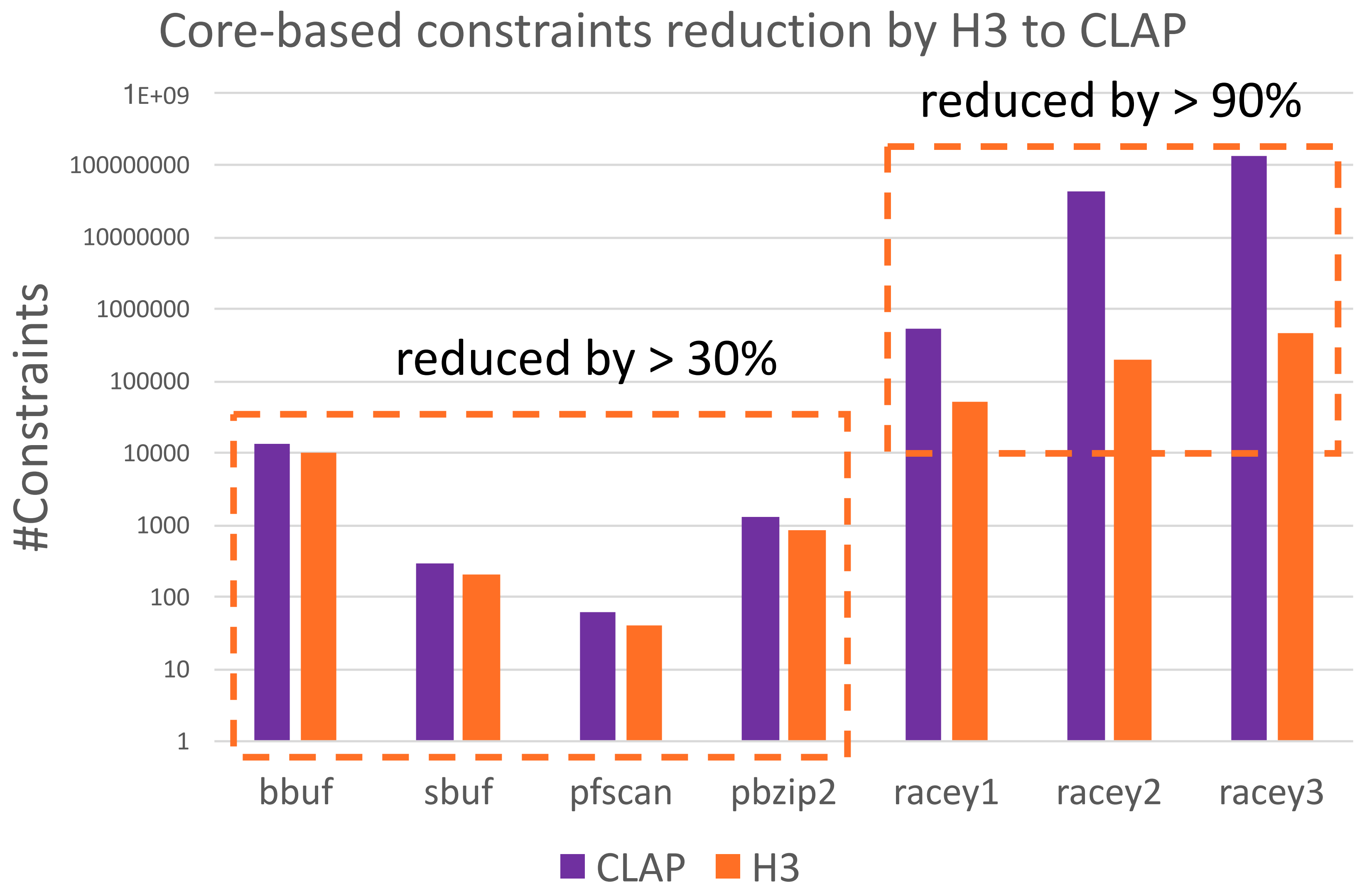


Runtime overhead

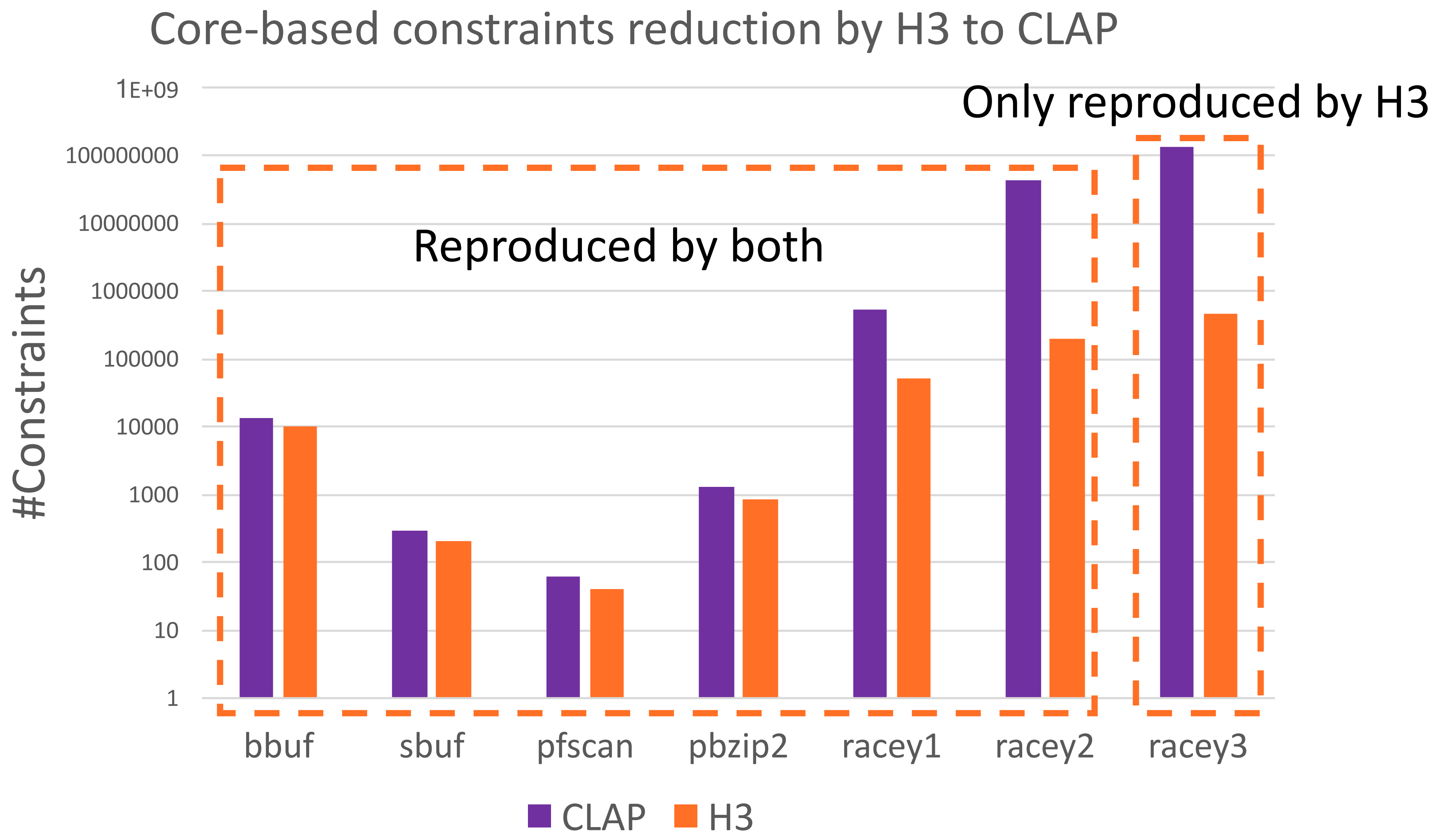


CLAP: 64.3% vs H3: 12.9%
reduction: 31.3%

Constraints reduction



Bug reproduction



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