Data Races vs. Data Race Bugs

by Baris Kasikci, Cristian Zamfir, and George Candea
École Polytechnique Fédérale de Lausanne (EPFL)

ASPLOS 2012

Christian Klauser

Data Races vs. Data Race Bugs

Telling the difference with "Portend"

- **Data race** defined as
  - two or more concurrent memory accesses to the same location
  - at least one access is a write-access
- "**fixing**" all data races rarely
  - desired (performance)
  - necessary (76%-90% are harmless)

Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Data race bugs are very hard to **reproduce**

To fixing data race bugs **proactively**:

- find data races
- determine whether harmful
- develop a fix

Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Data race bugs are very hard to **reproduce**

To fixing data race bugs **proactively**:

▶ find data races	▶ tools
▶ determine whether harmful	▶ this paper
▶ develop a fix	▶ by hand

Data Races vs. Data Race Bugs

Telling the difference with “Portend”

Key contributions

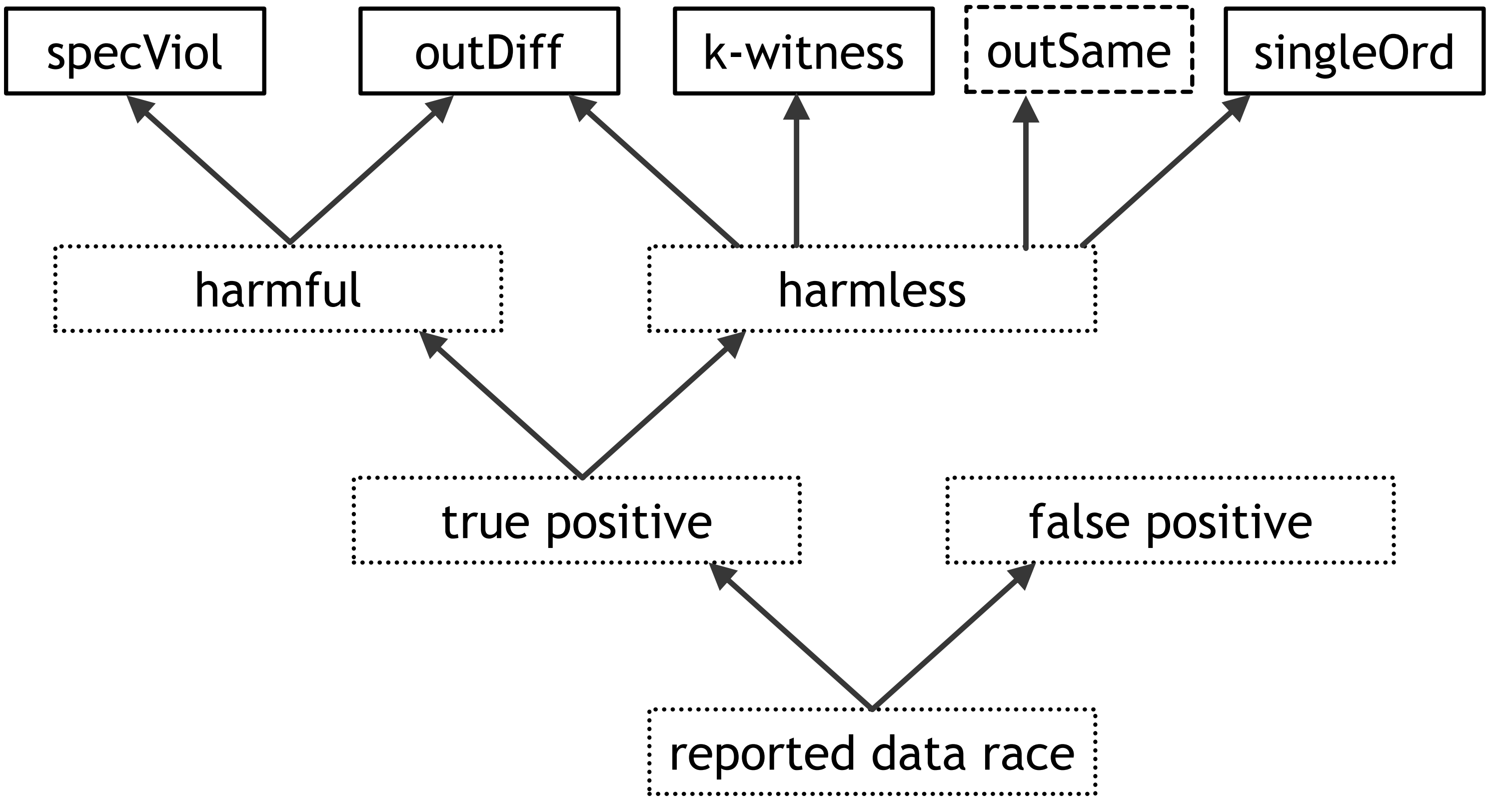
- Four-category **taxonomy** of data races
- Technique for **predicting consequences** of data races
- **Implementation** of the above: “Portend”

Data Race Classification

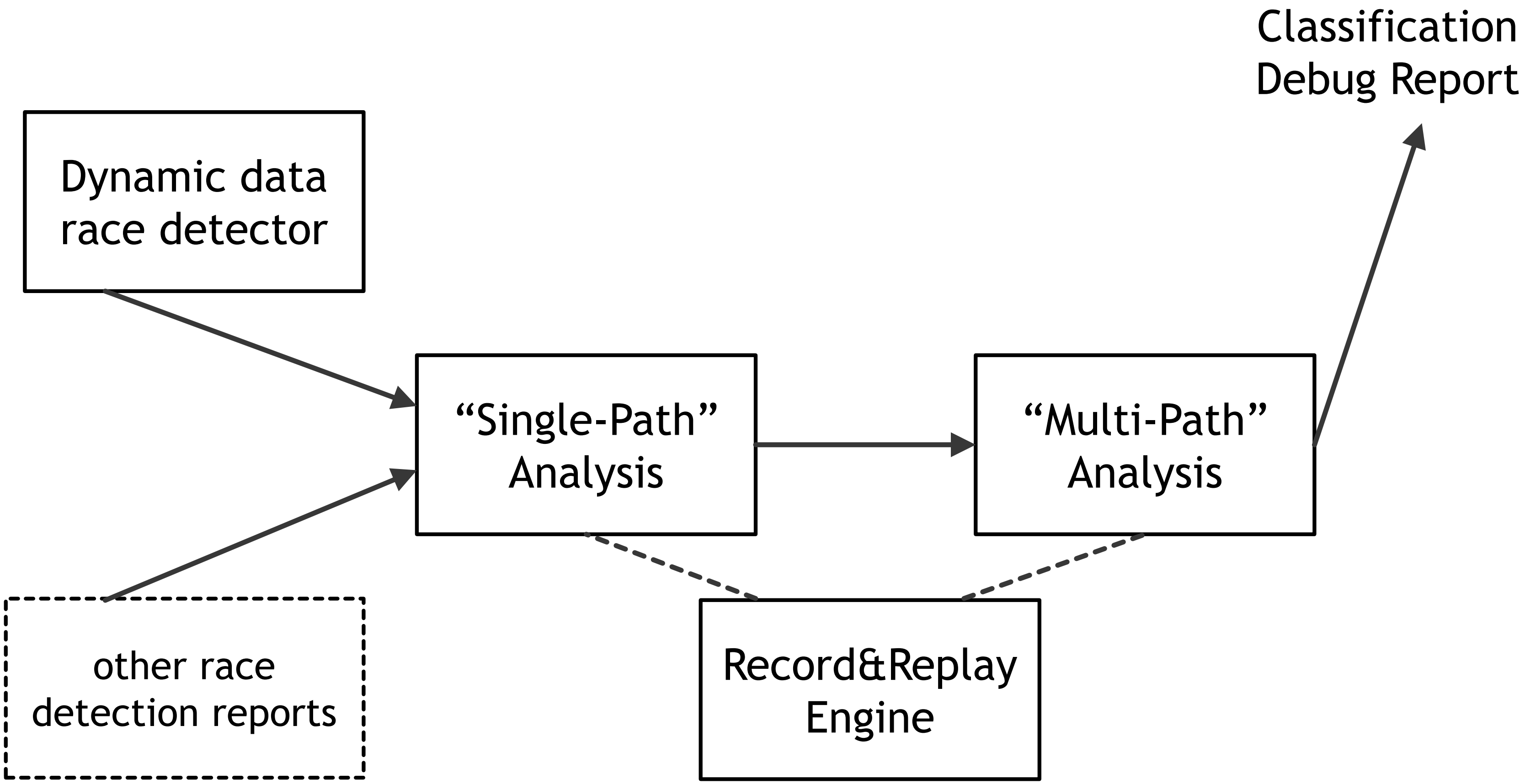
specViol	Program crashes or programmer specification violated
outDiff	Output differs between executions
k-witness	Harmless for at least *k* different paths and schedules
singleOrd	Only a single ordering possible[1]

[1] Arguably not a data race

Data Race Classification



Portend Pipeline



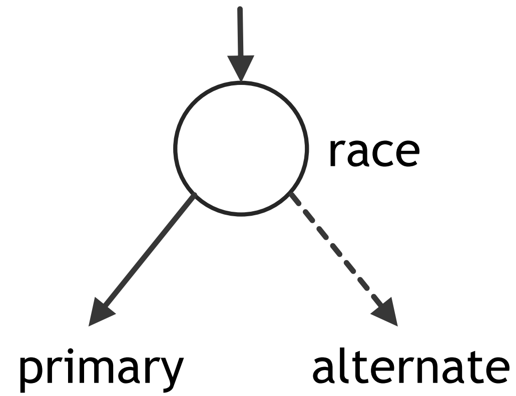
Single-Pre/Single-Post Analysis

Detect **singleOrd** data races, **deadlocks**

- Run **schedule** from report/detector
- On **first racing access** of data race d
  - Take **snapshot** pre of current program state
  - Continue
- On **second racing access** of data race d
  - Take **snapshot** $post$ of current program state

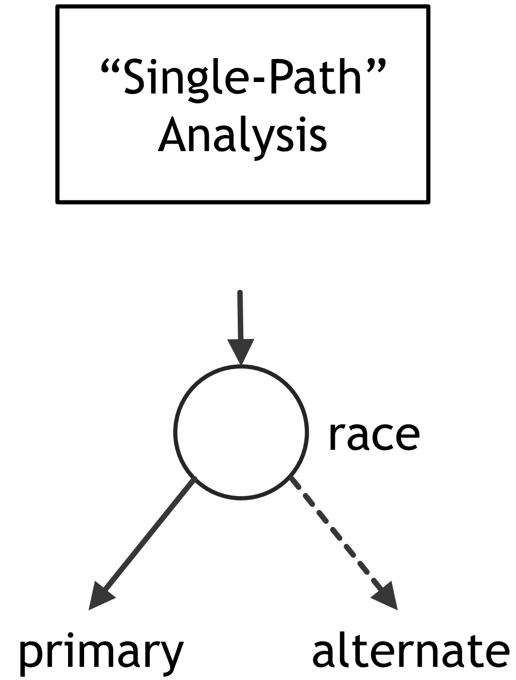
(cont)

“Single-Path” Analysis



Single-Pre/Single-Post Analysis

Detect **singleOrd** data races, **deadlocks**



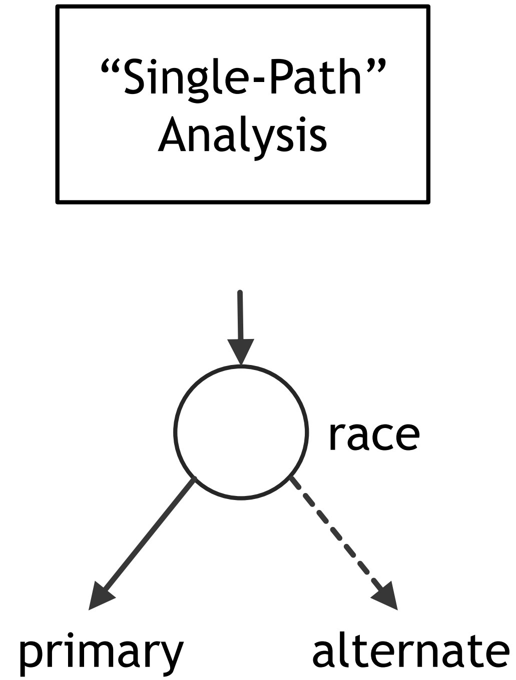
(cont)

- **Reset** program state to pre
- Run **different** schedule
  - **Preempt** thread that won last time
  - Capture program output as $\boldsymbol{alternate}$
- Run **original** schedule
  - **Reset** program state to $post$
  - Capture program output as $\boldsymbol{primary}$

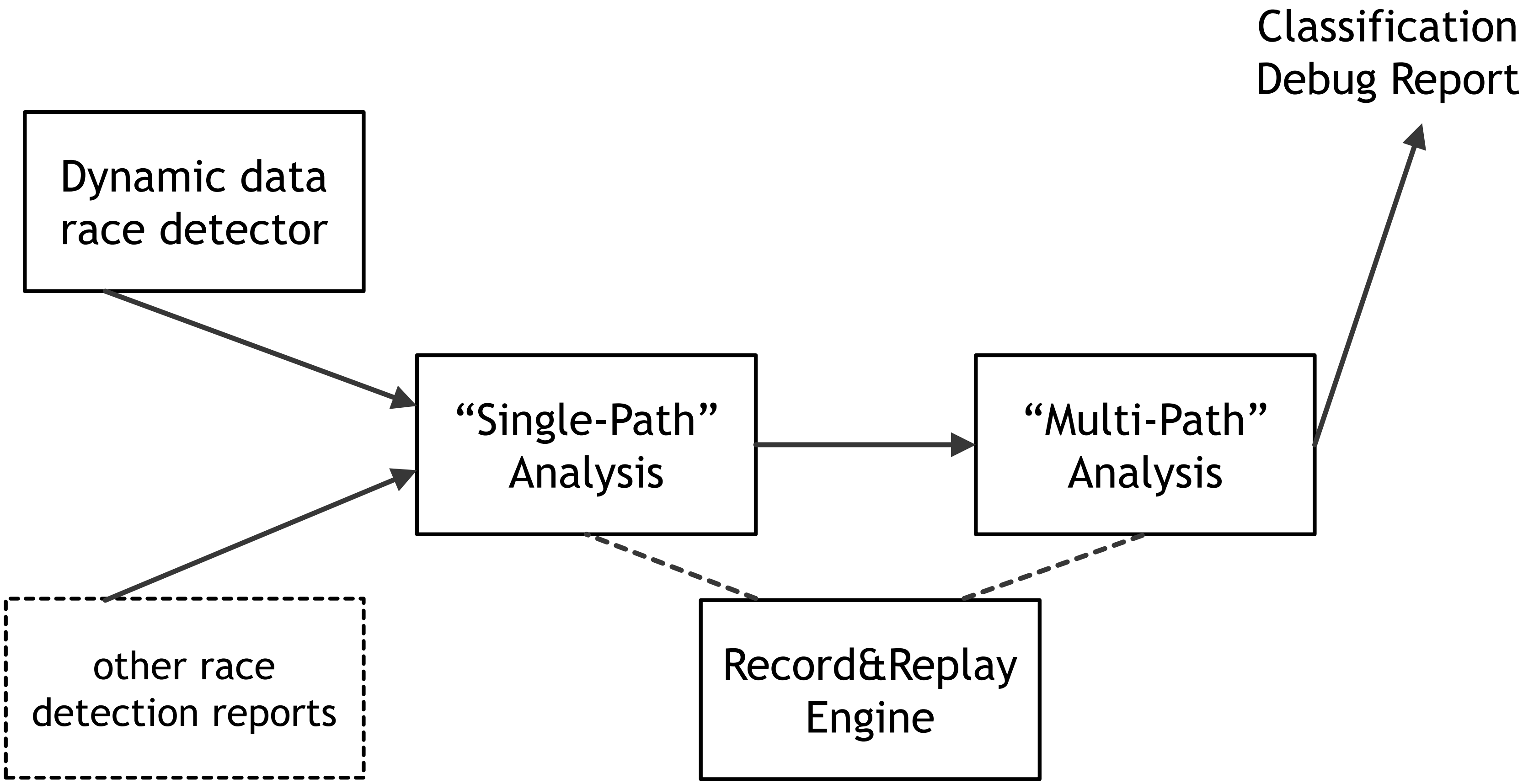
Single-Pre/Single-Post Analysis

Possible outcomes

- No alternate scheduling
  - detected deadlock, treat as **specViol**
  - program doesn’t terminate (**timeout**)
    - if **infinite loop**, treat as **specViol**
    - else treat as **singleOrd**
- Found alternate scheduling
  - check for semantic specification violations (**specViol**)
  - compare program outputs (**outDiff**, **outSame**)
    - important: compare program **output**, **not state** after race



Portend Pipeline



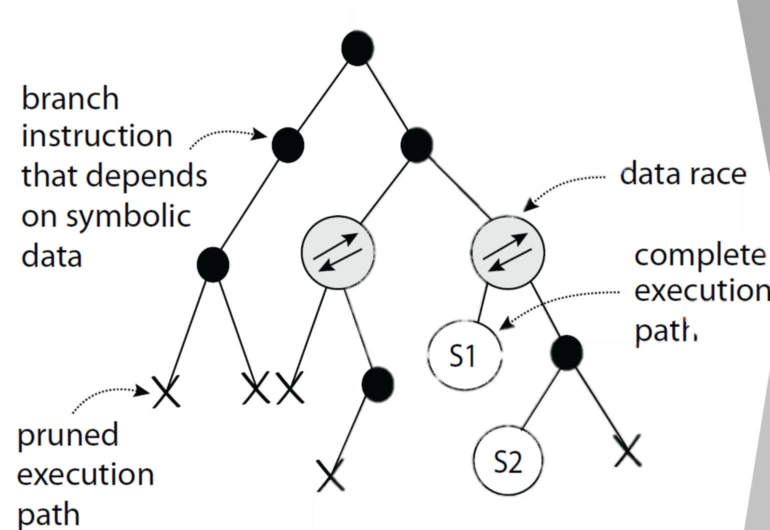
Multi-Path Data Race Analysis

Multiple inputs over the same thread schedule

“Multi-Path” Analysis

- Execute program using **symbolic** inputs
  - path explosion
  - Abandon paths that contradict thread schedule
- After 2nd racing access
  - collect M_p different paths
  - call these our “primaries”

(cont)



Multi-Path Data Race Analysis

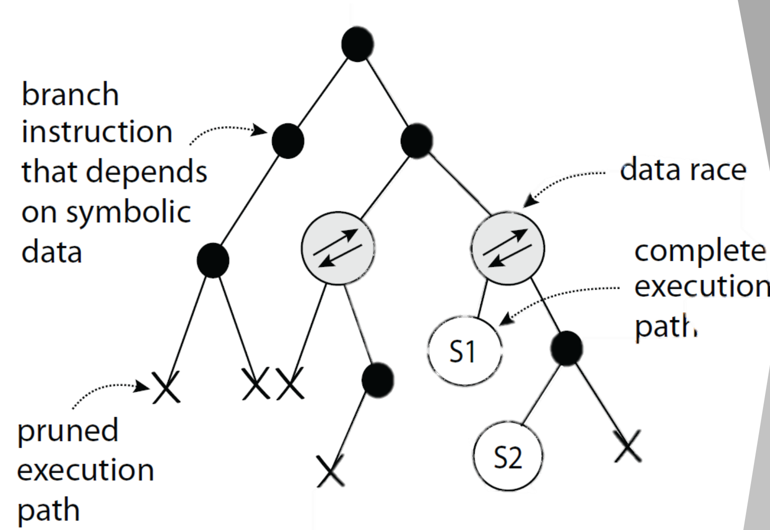
Multiple inputs over the same thread schedule

“Multi-Path” Analysis

(cont)

For each “primary”:

- record **symbolic output**
- let SMT solver find **concrete inputs**
- run **alternate scheduling** with concrete inputs
  - as in single-path analysis
- **compare** recorded output with symbolic reference output from primary
  - again via SMT solver



Multi-Schedule Data Race Analysis

Multiple schedules over multiple inputs

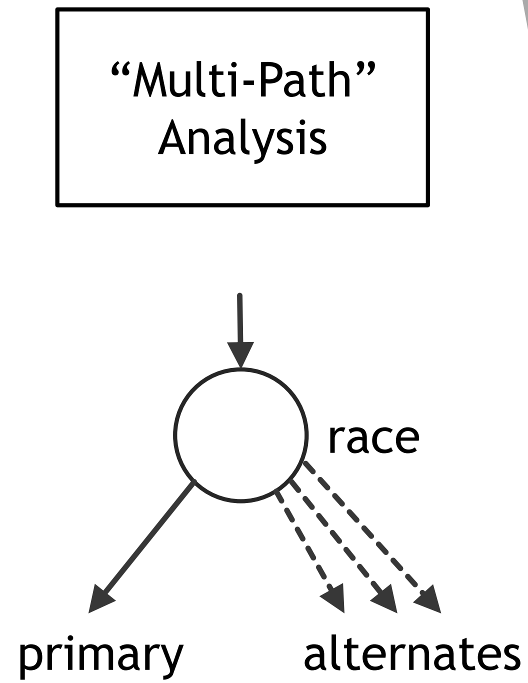
Idea: after data race, **don't follow existing schedule**

- Generate M_a **different** post-race schedules (**randomized**)
- **Record** output and **compare** with primary

If output **matches**, we have

$$k = M_p \times M_a$$

"**witnesses**" that race is harmless (**k-witness**)

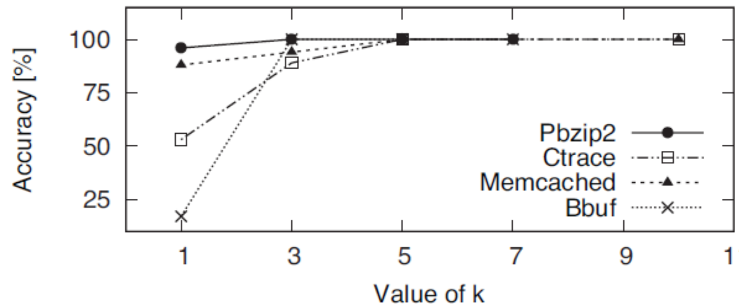


Portend Implementation Details

- Consumes **LLVM bitcode** programs
- 8K lines of **C++**, excluding libraries
  - **KLEE**, symbolic virtual machine for LLVM bitcode
  - **Cloud9** (EPFL), parallel symbolic execution engine
  - Symbolic **POSIX** emulation (part of Cloud9)
- Preemption points for single processor scheduling
  - POSIX synchronization primitives
  - data racing memory accesses

Portend Accuracy

How many races does Portend **classify correctly**?



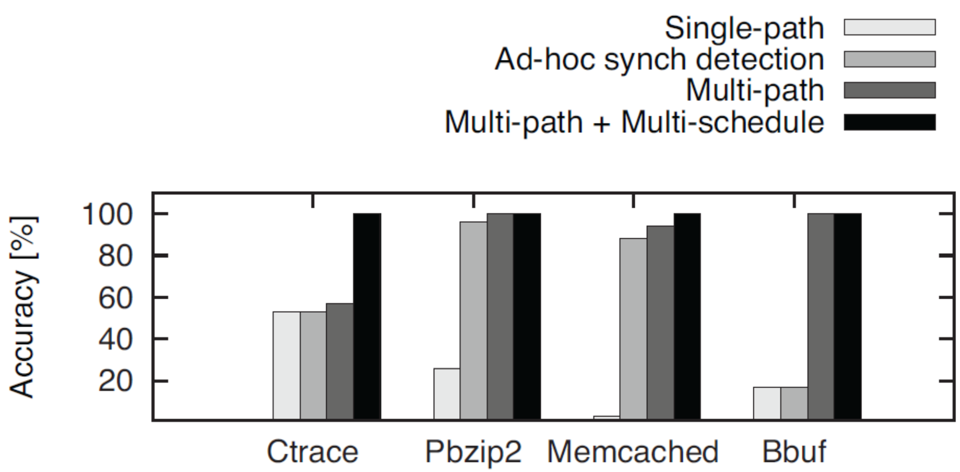
- Testsuite with 93 data races
  - SQLite 3.3.0, memcached 1.4.5, …
  - hand-written micro-benchmarks
- Classify data races by hand
- Compare with Portend classification
  - 92 out of 93 races correctly classified
  - One k-witness race was actually harmful
  - $k = 5$ (same result with $k = 10$)

Portend Results

What did Portend report?

Program	Total # of races	# of "Spec violated" races		
		Deadlock	Crash	Semantic
SQLite	1	1	0	0
pbzip2	31	0	3	0
ctrace	15	0	1	0
fmm	13	0	0	1
memcached	18	0	1	0

- Multi-path + multi-schedule were vital for accuracy
- **Single-path** often classified 80% or more data races as **singleOrd**
- **"Ctrace"** mainly resulted in **outDiff**



Portend Performance

- k-witness races take very long
  - many executions to simulate
- memcached: **11 minutes** to classify races
- Classification compared to bitcode interpretation
  - factor $\times 1.1$ to $\times 49.9$ longer
- Extreme ($\times 3$) variance in some cases

Limitations of Portend

- Performance nowhere near “interactive”
  - but $M_p \times M_a$ executions could be run in parallel
- Only POSIX synchronization primitives
  - Ignores machine-specific mechanisms (x86)
- Only single-processor model
  - Assumes memory-consistency, serializable execution

Thank you

Questions?

Additional Slides

Portend Performance Results

Program	Cloud9 running time (sec)	Portend classification time (sec)		
		Avg	*Min*	*Max*
SQLite	3.10	4.20	4.09	4.25
ocean	19.64	60.02	19.90	207.14
fmm	24.87	64.45	65.29	72.83
memcached	73.87	645.99	619.32	730.37
pbzip2	15.30	360.72	61.36	763.43
ctrace	3.67	24.29	5.54	41.08
bbuf	1.81	4.47	4.77	5.82
AVV	0.72	0.83	0.78	1.02
DCL	0.74	0.85	0.83	0.89
DBM	0.72	0.81	0.79	0.83
RW	0.74	0.81	0.81	0.82

References

- Illustrations and tables taken from
“Data Races vs. Data Race Bugs: Telling the Difference with Portend, Baris Kasikci, Cristian Zamfir, and George Candea, ASPLOS’12”