RaceMob: Crowdsourced Data Race Detection

Baris Kasikci, Cristian Zamfir, George Candea

Presented By:
Islam Harb
2014

Agenda

- Motivation
- Data Race Detection Classes
- RaceMob
- Implementation
- Evaluation

Motivation
(The Problem?)

- Data races as a problem of the concurrency.

- Data races are represented in
  - Atomicity (e.g. access same memory location at same time).
  - Order violation (e.g. bad pointers).

- Difficult to discover. Usually requires significant overhead.

Few is Many

- Although only 5-24% of data races have harmful effect(s), their consequences were Catastrophic.

- If I am a top coder, why would I worry?
  - C/C++ standards allow compilers' optimization that might lead to data races.

- Therefore, data race detectors are highly recommended.

Static Data Race Detection

- **Static Detection:** Analyze the code without execution. (Reasoning)
- Pros:
  - Offline (No runtime overhead).
  - Fast and Scale to large code bases.
- Cons:
  - False Positives (unreal data races).

Dynamic Data Race Detection

- **Dynamic Detection:** Monitor memory access and synchronization at *runtime*.
- Pros:
  - More accurate (very low FPs rates).
- Cons:
  - Test Cases depended. Miss data races that aren’t seen during execution (False Negative)
  - runtime overhead.

RaceMob

- Combines static and dynamic detections to obtain both accuracy and low runtime overhead.

- RaceMob is a three-phased detector.
  - First, static detection phase (potential races with few false negatives).
  - Dynamic phase.
  - Crowdsources the validation phase to users machines.

Static RaceMob [Phase I]

- The static phase of the RaceMob is done via the RELAY.
- RELAY is a “lock-set” data race detector.
- Data race is flagged when:
  - At least two accesses to memory locations that are the same or may alias.
  - One of the accesses is write.
  - The accesses are not guarded by at least one common lock.
- Based on RELAY report, RaceMob instruments all suspected memory access and synchronization operations.

Dynamic RaceMob [Phase 2]

- The Dynamic phase of the RaceMob.
- The hive instructs and distributes the validation task through the users sites.
- Dynamic phase itself is consisted of there phases:
  1. DCI: Dynamic Context Inference [Always ON].
  2. On-Demand Data Race Detection [ON/OFF].
  3. Schedule Steering [ON/OFF].

DCI: Dynamic Context Inference

- Looks for concrete instances at runtime at the users machines.
- The concrete instances should validate the candidate data race and confirm on whether the racing accesses are made by two different threads.
- DCI, keeps track of addresses of potential racing accesses and the Thread's ID.
- Negligible runtime overhead (0.01%), there feasible to be always ON.

On-Demand Data Race Detection

- **Starts** tracking the happens-before relationships **once** first potential racing access is made.
- **Stops** tracking:
  - “happens-before” occur between first accessing thread and all other threads. [No Race]
  - Second racing access occur before such “happens-before”. [True Race]

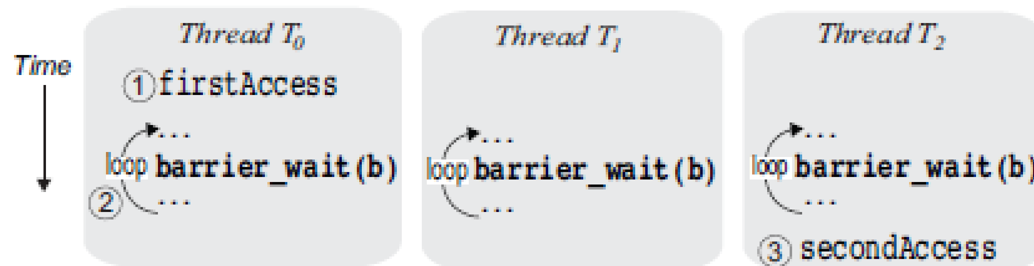


Figure 3: Minimal monitoring in DCI: For this example, DCI stops tracking synchronization operations as soon as each thread goes once through the barrier.

Schedule Steering

- Hive instructs one of the orders ("primary" or "alternative") to be validated.
- RaceMob may pause the accessing thread with "wait" operation to enforce the intended order.

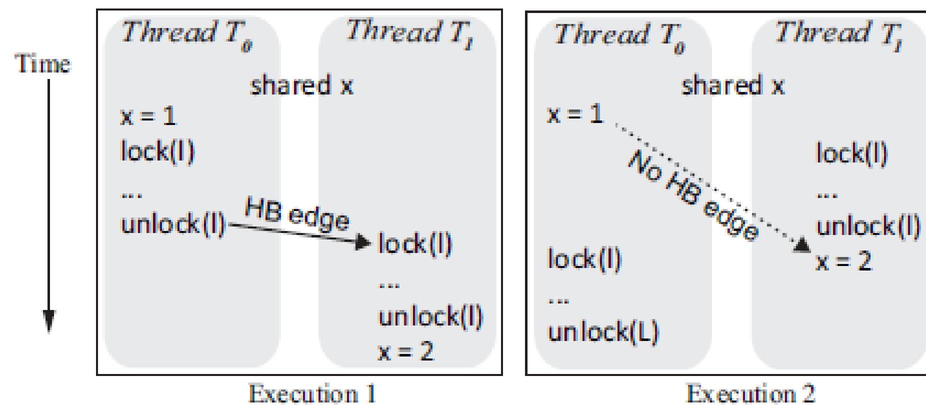


Figure 1: False negatives in happens-before (HB) dynamic race detectors: the race on x is not detected in Execution 1, but it is detected in Execution 2.

Crwodsourcing Overview [Phase 3]

- Crowdsourcing the validation.

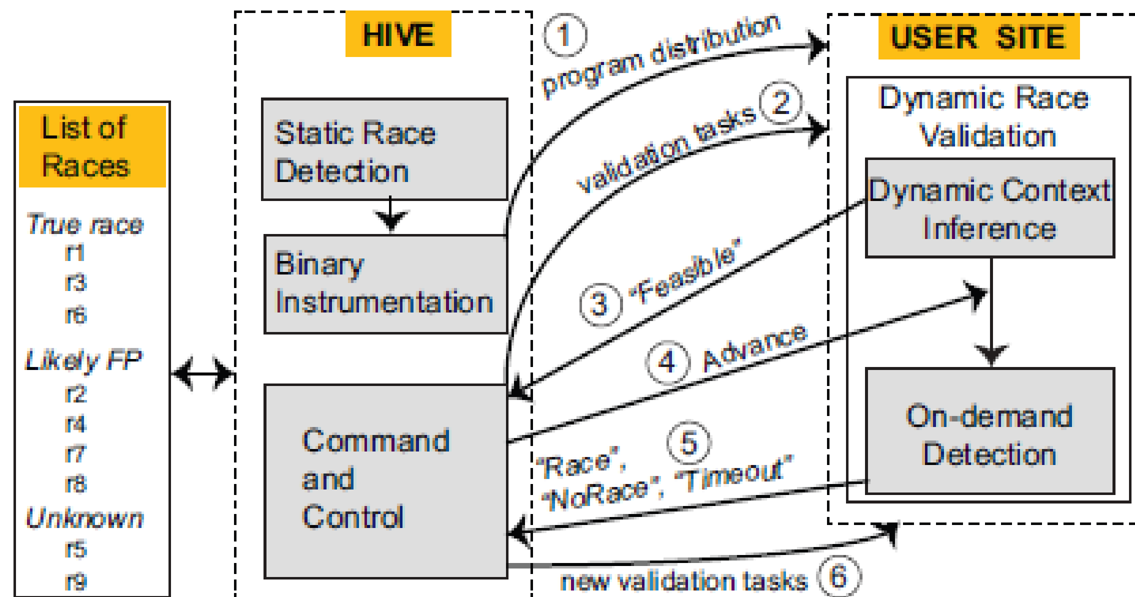
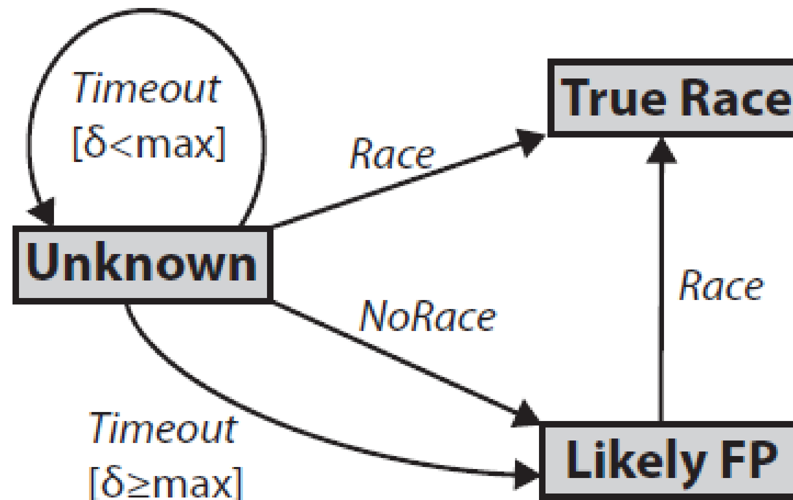


Figure 2: RaceMob's crowdsourced architecture: A static detection phase, run on the hive, is followed by a dynamic validation phase on users' machines.

RaceMob: Reaching Verdict

- True Race is definite.
  - Should get a proof from any of the user-sites!
- Likely False Positive is probablisitic.
  - The more “No Race” & “Timeout” reports, the more probability that it is False Positive.



Implementation

- 4,147 C++ Lines of Code.
- 2, 850 Python – Hive and user-side daemon.
- Used C++11 weak atomic store/load operations.
- Hive is based on LLVM

Empty Loop Optimization

- Empty loop bodies caught and suspected as a data race candidate: `While(notDone){}`
  - Not instrumented.
  - Reported directly to the developer by the hive.
  - Never reach to the user-sites for further validation.
  - Otherwise, excessive overhead encounters.

Evaluation

- Does it work on Real Code (Real Applications)?
- Efficient?
- RaceMob vs. state-of-the-art?
- Scale with No. of threads?

Test Environment

- Small scale real deployment on Authors laptops.
  - Thinkpad Laptops, Intel 2620M Processors, 8 GB RAM, Ubuntu Linux 12.04.
- 1, 754 simulated users sites.
- Test Machines:
  - 48-core AMD Opteron 6176 (2.3 GHZ), 512 GB RAM, OS: Ubuntu Linux 11.04 [Simulated Users]
  - Two 8-core Intel Xeon E5405, 20 GB RAM, OS: Ubuntu 11.10 [Hive + Simulated Users]

Applications

- SQLite
- Bzip2
- Memcached
- Ocean
- Fmm
- Barnes
- Apache
- Others

Evaluation

- ~13% (106) True Race. [don’t forget: Few is Many!]
- 77% are Likely FP
- No False Negative.

Program		Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
Size (LOC)		138,456	113,326	19,397	9,126	7,580	6,551	3,521	3,586	2,053	2,033
Race candidates		118	88	7	176	166	115	65	65	24	17
True Race	Causes hang	0	3	0	0	0	0	0	0	0	0
	Causes crash	0	0	0	0	0	0	3	0	0	0
	Both orders	0	0	1	5	10	0	2	0	0	0
	Single order	8	0	0	53	6	3	4	2	4	2
Likely FP	Not aliasing	10	31	0	33	65	13	0	18	2	0
	Context	61	10	2	61	28	42	21	28	10	4
	Synchronization	1	37	3	10	49	47	34	13	7	11
Unknown		38	7	1	14	8	10	1	4	1	0

Overall Overhead

- *Less runtime overhead.*
- Static Stage is Offline ~3 minutes for all programs, except for Apache and SQLite ~ less than 1 hour.

Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
1.74	1.60	0.10	4.54	2.98	2.05	2.90	1.27	3.00	3.03

Table 2: Runtime overhead of race detection as a percentage of uninstrumented execution. Average overhead is 2.32%, and maximum overhead is 4.54%.

Instrumentation vs. Validation

- Overhead = Instrumentation + Validation

-Instrumentation overhead is negligible with respect to the Validation overhead

-DCI is negligible ~0.1%

- Dynamic Data Race is the black portion. [Lion Share]

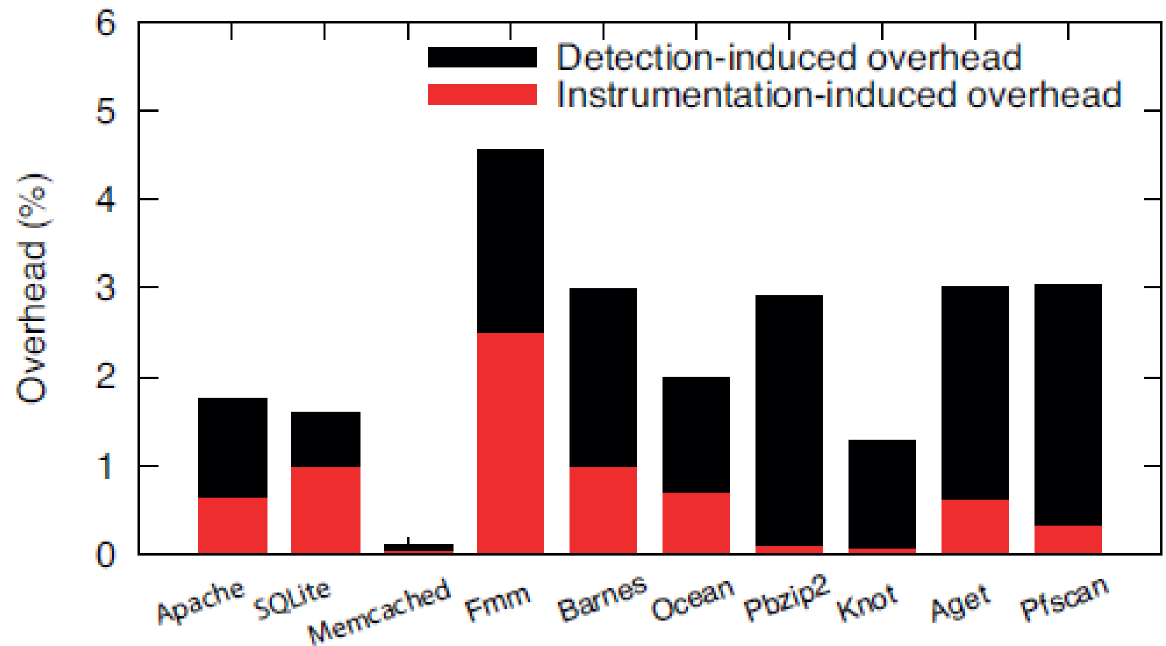


Figure 5: Breakdown of average overhead into instrumentation-induced overhead and detection-induced overhead.

Evaluation

Comparison State-of-the-Art

- RaceMob, RELAY and TSAN
- RaceMob detected 4 extra True Races than TSAN

Program	Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
RaceMob	8	3	1	58	16	3	9	2	4	2
TSAN	8	3	0	58	16	3	9	2	2	1
RELAY	118	88	7	176	166	115	65	157	256	17

Table 3: Race detection results with RaceMob, ThreadSanitizer (TSAN), and RELAY. Each cell shows the number of reported races. The data races reported by RaceMob and TSAN are all true data races. The only true data races among the ones detected by RELAY are the ones in the row "RaceMob". To the best of our knowledge, two of the data races that cause a hang in SQLite were not previously reported.

Comparative Overhead

Program	Aggregate overhead with RaceMob [# of race candidates × # of users] in %	TSAN user-perceived overhead in %
Apache	339.30	25,207.79
SQLite	281.60	1,428.57
Memcached	2.20	3,102.32
Fmm	1,598.08	47,888.07
Barnes	989.36	30,640.00
Ocean	360.70	3,069.39
Pbzip2	377.00	3,001.00
Knot	165.10	751.47
Aget	144.00	184.22
Pfscan	103.20	13,402.15

Table 4: RaceMob aggregate overhead vs. TSAN's average overhead, relative to uninstrumented execution. RaceMob's aggregate overhead is across all the executions for all users. For TSAN, we report the average overhead of executing all the available test cases.

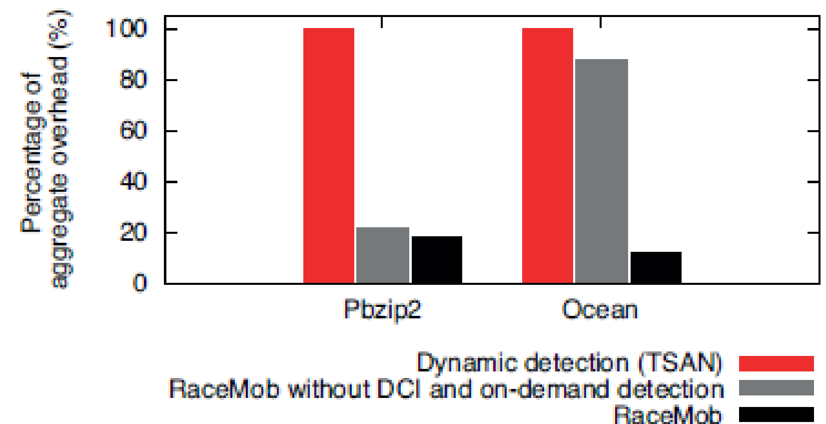


Figure 6: Contribution of each technique to lowering the aggregate overhead of RaceMob. Dynamic detection represents detection with TSAN. RaceMob without DCI and on-demand detection just uses static data race detection to prune the number of accesses to monitor.

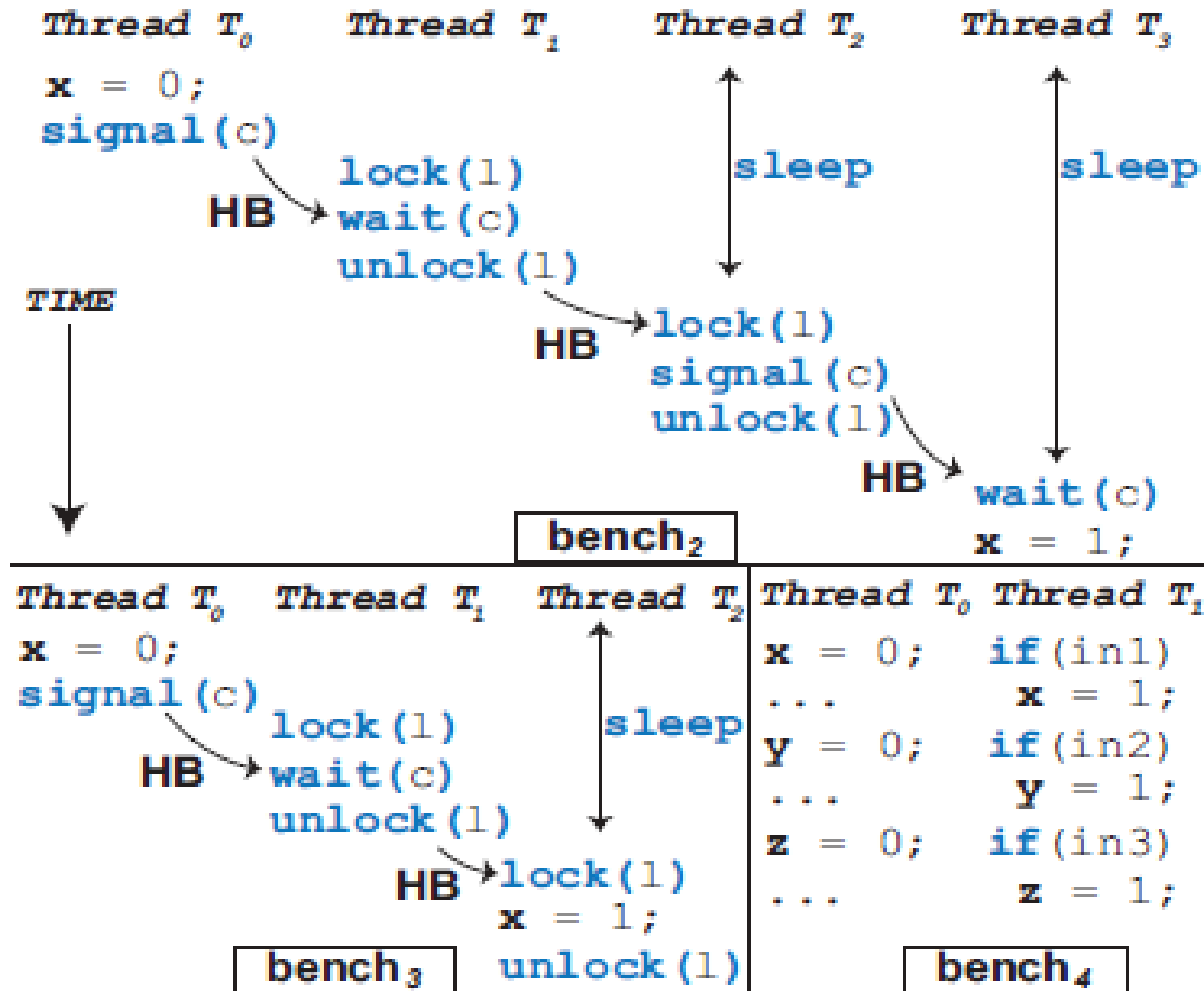
Schedule Steering is Significant

- RaceMob's Schedule Steering plays very important role.

- SQLite & Pbzip2:
  - When NOT instrumented – 10,000 executions but no "hang".
  - When instrumented (SS is ON) – 3 hangs in 176 executions.

- Pbzip2:
  - When NOT instrumented – 10,000 executions but no "crash".
  - When instrumented (SS is ON) – 4 crashes in 130 executions.

Concurrency Testing Tools



Concurrency Testing Tools(continued)

Evaluation

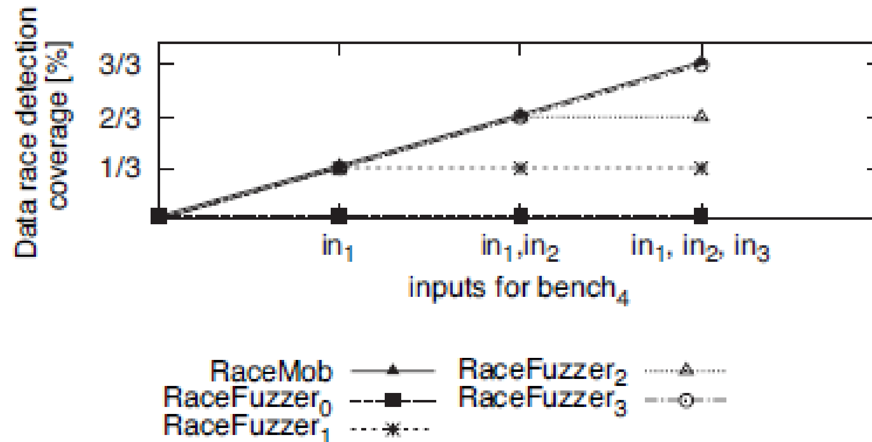


Figure 8: Data race detection coverage for RaceMob vs. RaceFuzzer. To do as well as RaceMob, RaceFuzzer must have a priori access to all test cases (the $RaceFuzzer_3$ curve).

Tool	$bench_1$	$bench_2$	$bench_3$	$bench_4$
RaceMob	1 / 1	1 / 1	1 / 1	3 / 3
RaceFuzzer	1 / 1	1 / 1	1 / 1	0 – 3 / 3
Portend	0 / 1	0 / 1	0 / 1	3 / 3

Table 5: RaceMob vs. concurrency testing tools: Ratio of races detected in each benchmark to the total number of races in that benchmark.

Big Size Problems

- How this affect on scalability?
  - 10 MB file – concurrent requests [Apache & Knot]
  - Insert, modify & remove 5,000 items from database & object cache [SQLite, Memcached]
  - Similarly, enlarge problem size in Ocean, Pbzip2 and Barnes.

Application Threads Scalability

- Scalability Experiment:
  - Varied threads No. from 2-32.
  - RaceMob runs on 8-core machine.

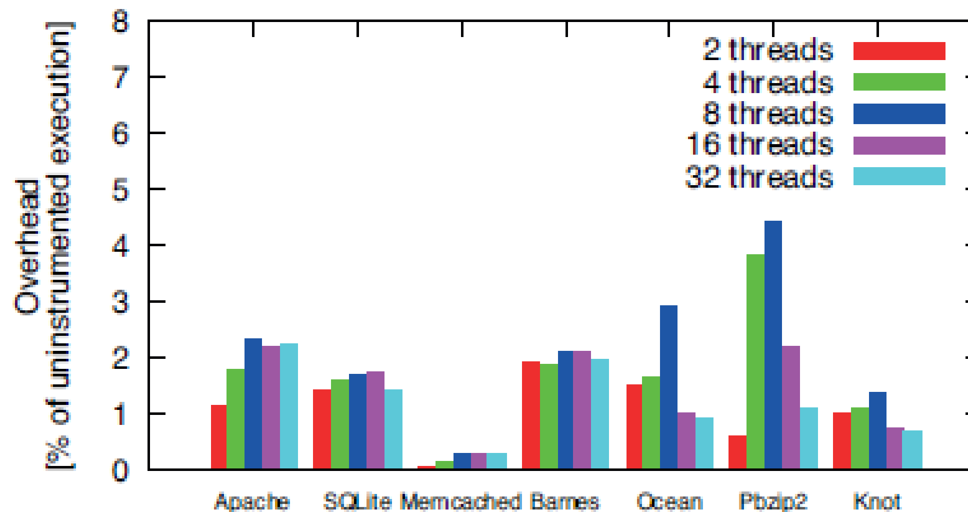


Figure 9: RaceMob scalability: Induced overhead as a function of the number of application threads.

Thanks!
Any Questions?