Accurate and Efficient Filtering for the Intel Thread Checker Race Detector

Paul Sack, Brian E. Bliss*, Zhiqiang Ma*,
Paul Petersen*, and Josep Torrellas
*University of Illinois and Intel Corporation**

October 21, 2006

ASID ‘06

i-acoma group

Background: What is a data race?

- 2 threads access a variable without *synchronization*, at least one write
- Accesses could have occurred in either order, leading to unexpected behavior

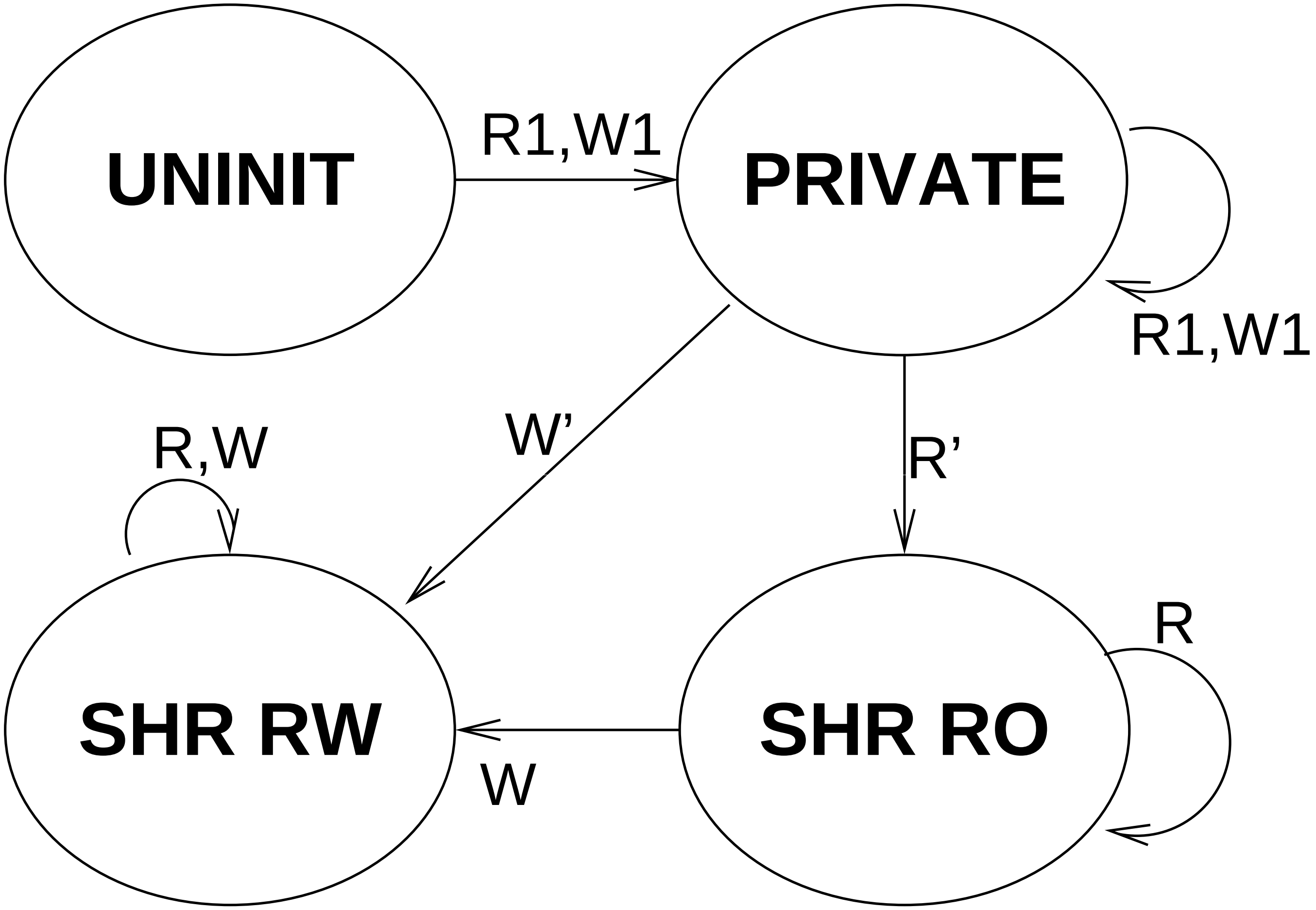
Background

- Data races difficult to debug
  - Rare
  - Difficult to reproduce
  - Sensitive to
    - Thread scheduling, interrupt timing, etc.
    - Debuggers, compiler optimizations
    - Which system used
- Data race detectors:
  - Lockset and vector-clock based

Background: Lockset-based Detectors

- Lockset: a set of common locks used to access a global variable (Eraser [Savage '97])
- Works by intersecting saved lockset of variable with current lockset of thread
- If intersection is null, data race!
  - Really *violation of locking discipline*
- What about variable initialization and thread-private data?

Background: Eraser FSM



UNINIT

R1,W1

PRIVATE

R1,W1

R,W

W'

R'

SHR RW

W

SHR RO

R

Background: Vector-clock Detectors

- E.g., RecPlay [Ronsse et al ’99], [Netzer and Miller ‘91]
- *Segments* are instructions executed between synchronization points
- Vector clocks order thread segments:
  - Used to tell if segment *x* on thread 1 executed before, in parallel, or after segment *y* on thread 2
- Race detected if current access to memory races with prior accesses:
  - At least one access must be a write
  - Both segments must be unordered

Background: How do they compare?

Lockset

- Faster?
- Only works with lock-based synch
  - Can catch some races that vector-clocks can't
- Detects *violations of locking discipline*

Vector-clock

- Slower?
- Works with all kinds of synchronization
- Provides more information
- Detects *races*

Background

- Race detectors find data races
  - Great!
- BUT: Either slow or limited

Background: Detectors

- Eraser [Savage *et al* ’97]
  - Efficient, but only really works with locks
  - Doesn’t tell user much
- RaceTrack [Yu *et al* ’05]
  - Efficient, only works with managed run-time code
  - Also doesn’t tell user much
- Most tell user:
  - Which variable
  - One PC
  - One or two thread IDs

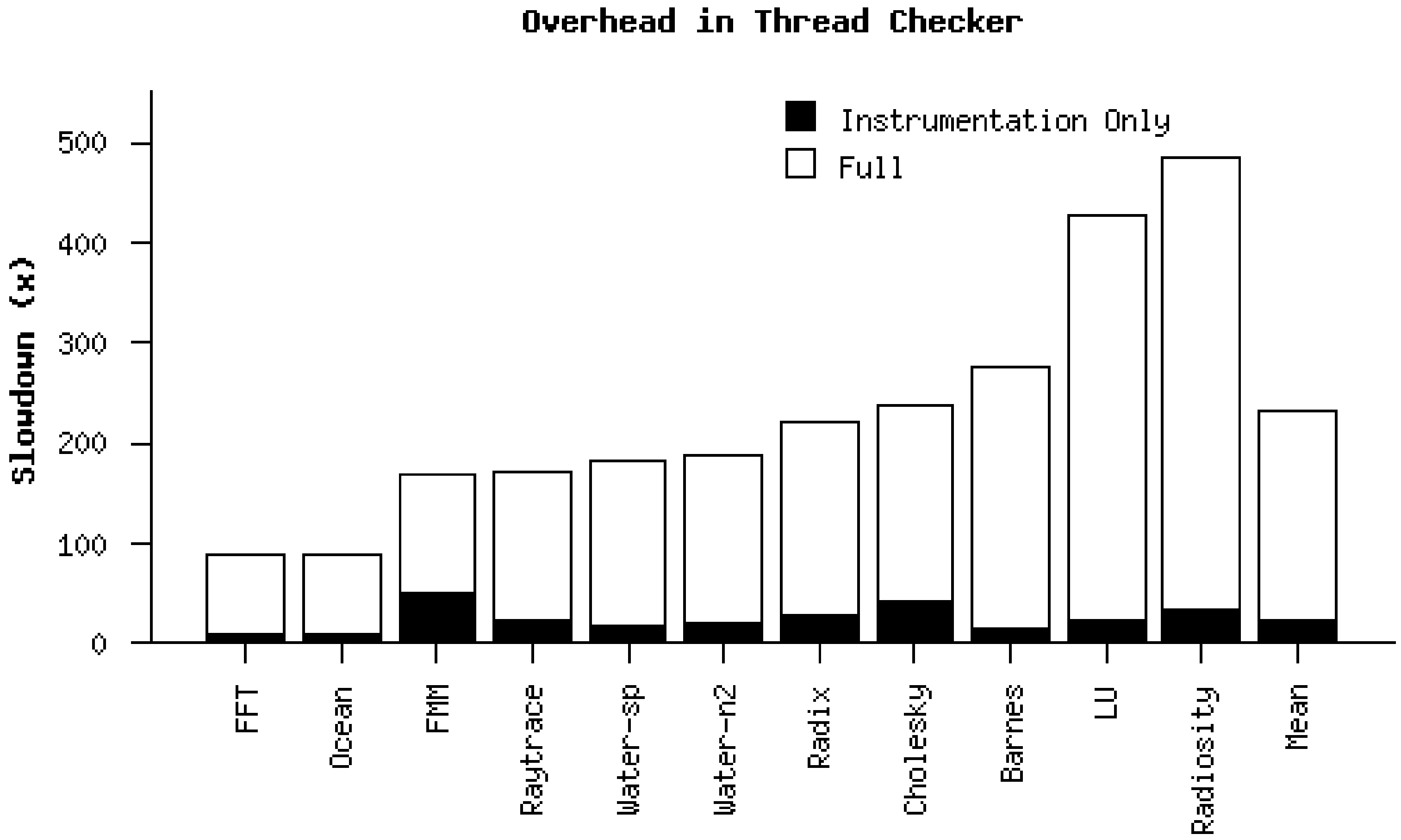
Background: Detectors

- Intel Thread Checker
  - Uses vector-clock algorithm
  - Works with any binary code (C, C++, Fortran, assembly)
  - Does not require recompilation or full source code
  - Reports thread IDs, source code locations, and *call stacks* for both racing accesses
  - *Extremely slow*

Outline

- Motivation
- Approach
- Evaluation
- Conclusion

Motivation


Overhead in Thread Checker
Slowdown (x)
0
100
200
300
400
500
■ Instrumentation Only
☐ Full
FFT
Ocean
FMM
Raytrace
Water-sp
Water-n2
Radix
Cholesky
Barnes
LU
Radiosity
Mean

Approach

- Many possibilities:
  - Improve instrumentation (e.g., Aliter [Gao ’05])
  - Optimize analysis
    - E.g., parallelize

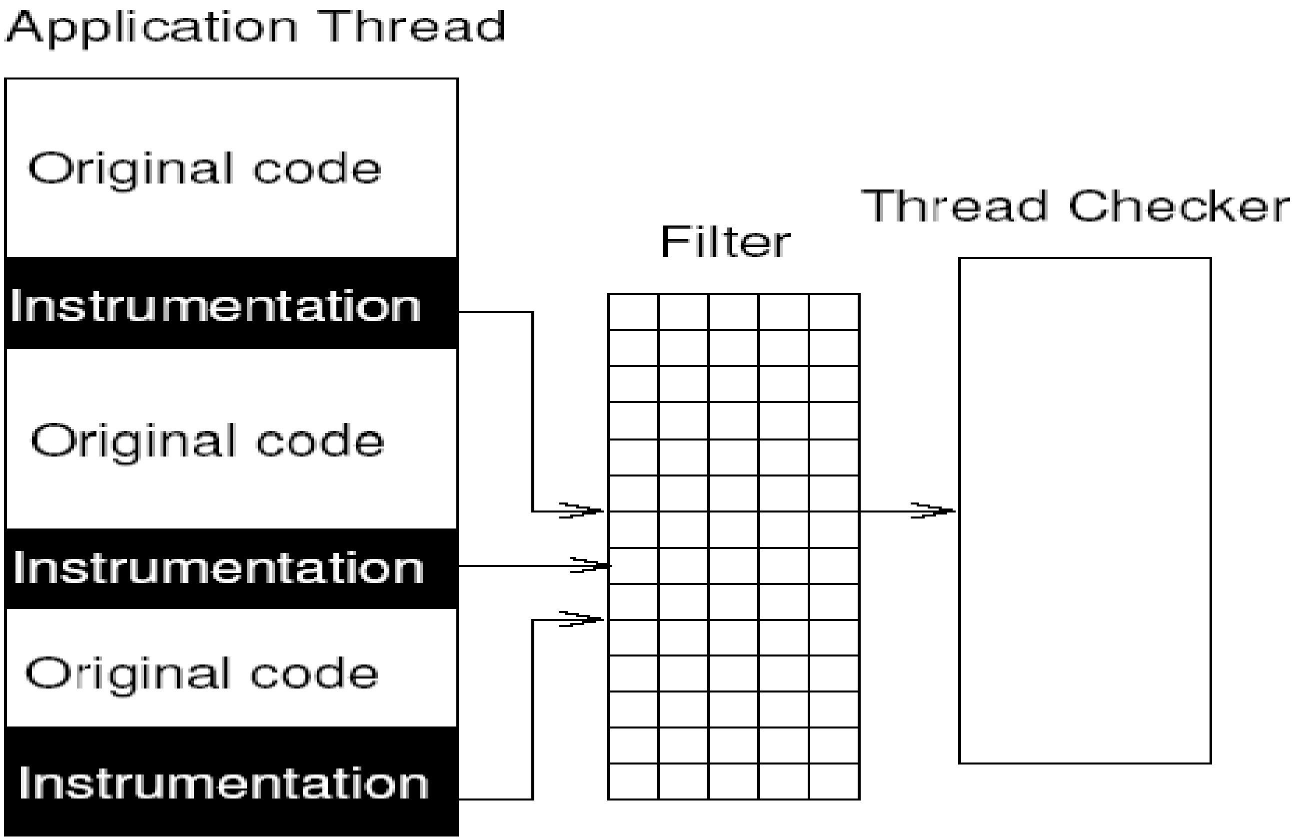
None of these are *general* approaches

Approach

- Observation: 99+% of accesses not involved in data races
  - For some classes of accesses, 100% not involved

Classic technique: Filtering

Approach


Application Thread
Original code
Instrumentation
Original code
Instrumentation
Original code
Instrumentation
Filter
Thread Checker

Approach: Stack filter

- Stack accesses not typically involved in data races
  - Compare address to stack base and limit; if in range, filter out
  - Add stage to check for *cross-thread* stack accesses to disable the filter
  - Very fast filter—just two comparisons

Approach: Duplicate filter

- Temporal locality exists within segments
- Equivalent accesses to the same address can be safely filtered
- E.g.,

```
Barrier();
X = 8;
X++;
Barrier();
```

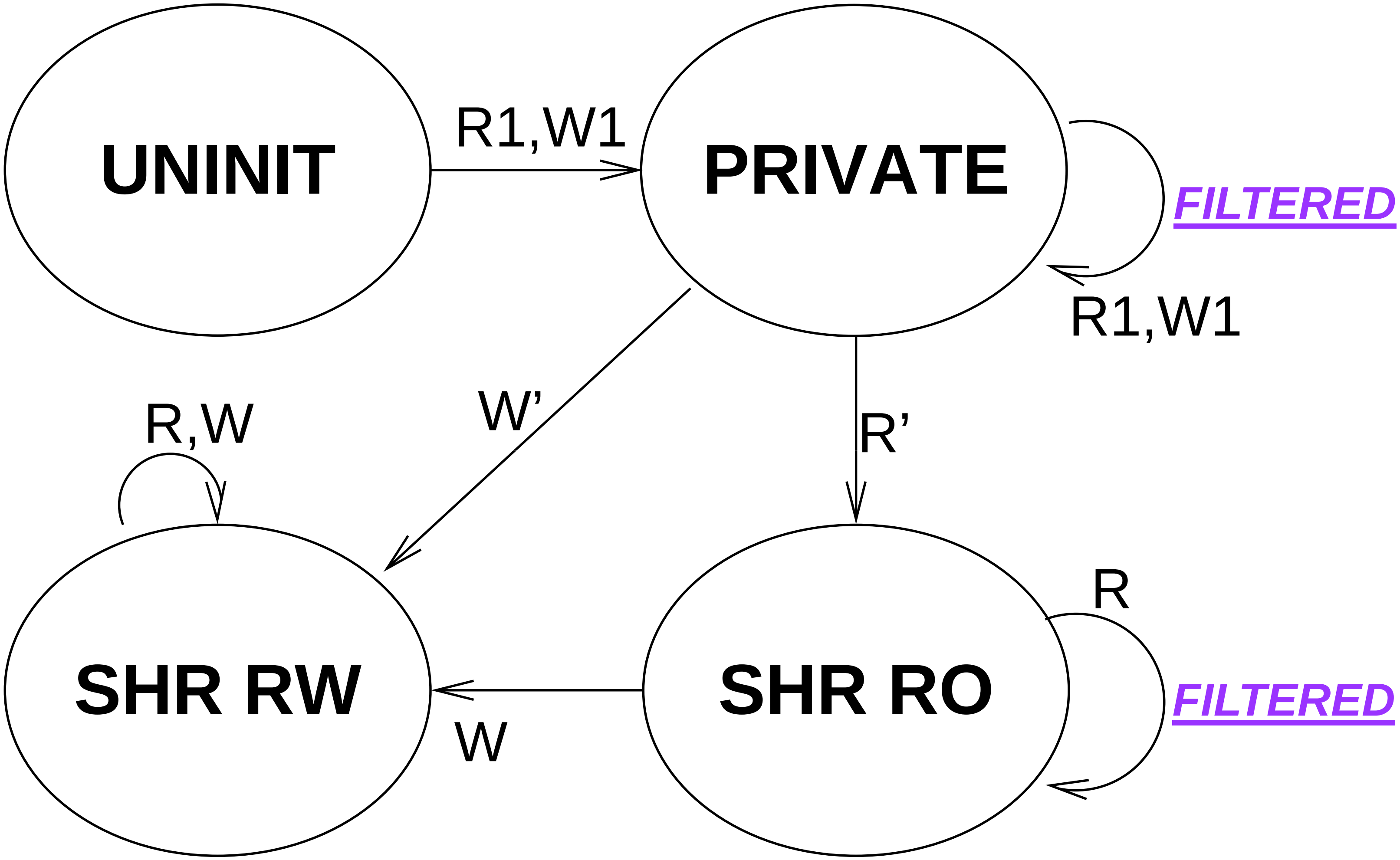
Approach: Duplicate filter

- Per-thread table keeps track of accesses within segment
  - 16k-entry, direct-mapped cache
  - Address, size, type of access
- Duplicate accesses filtered
- Feedback loop: variables involved in data race marked in table, not filtered
  - Provides user with more information
- Relatively-fast: private table lookup

Approach: FSM filter

- Premise: Many accesses to non-stack variables are thread private or shared read-only
- Use Eraser state machine

Approach: FSM Filter



Approach: FSM Filter

- Only filter which *can* lose data races
  - Uses past to predict future
  - But many data races repetitive
- Optional
- Slower—global table with some synchronization
- Evaluation will see if it is worthwhile

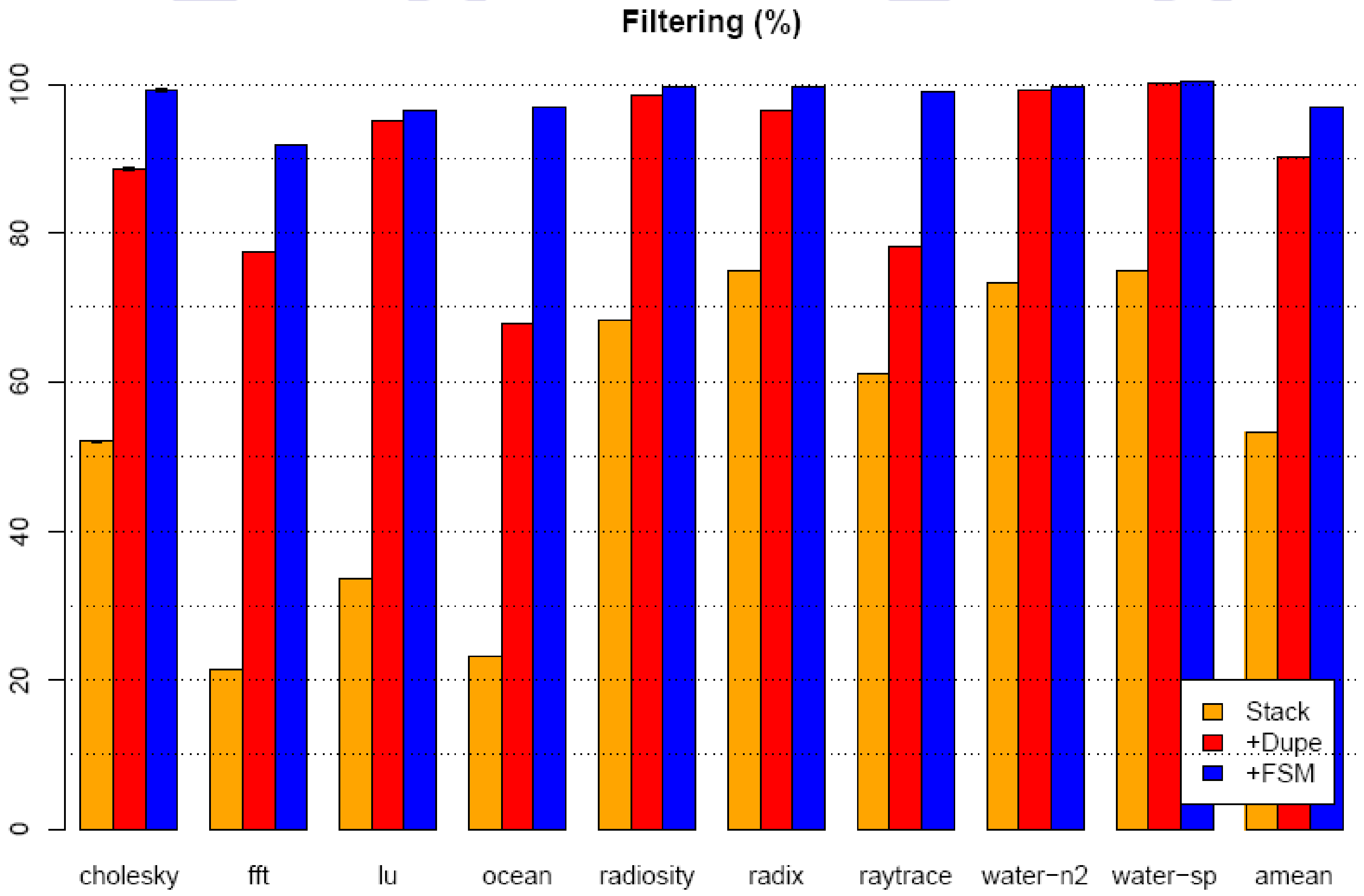
Approach: FSM Filter

- Global 4M entry, 4-way table, LRU replacement
- Each entry covers one 32-bit word
- Contains:
  - FSM state
  - Thread ID
  - Tag
- Table takes up 4MB storage
- Fast-path optimizations avoid synch

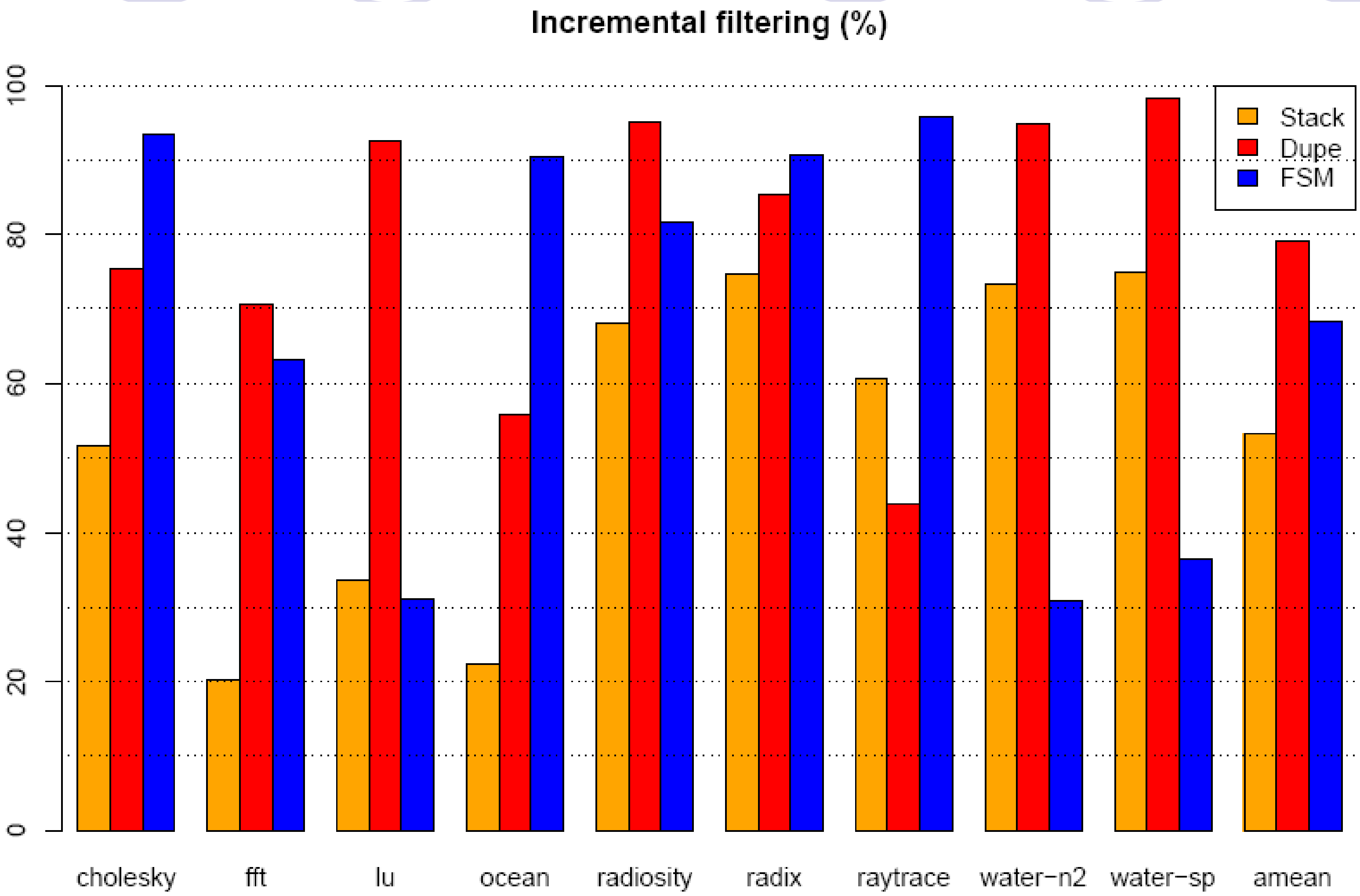
Evaluation: Methodology

- 9 Splash-2 apps
- Standard data set
- 4-way Pentium system
- Performance measurements repeated 9x
- Filter rate measurements repeated 3x

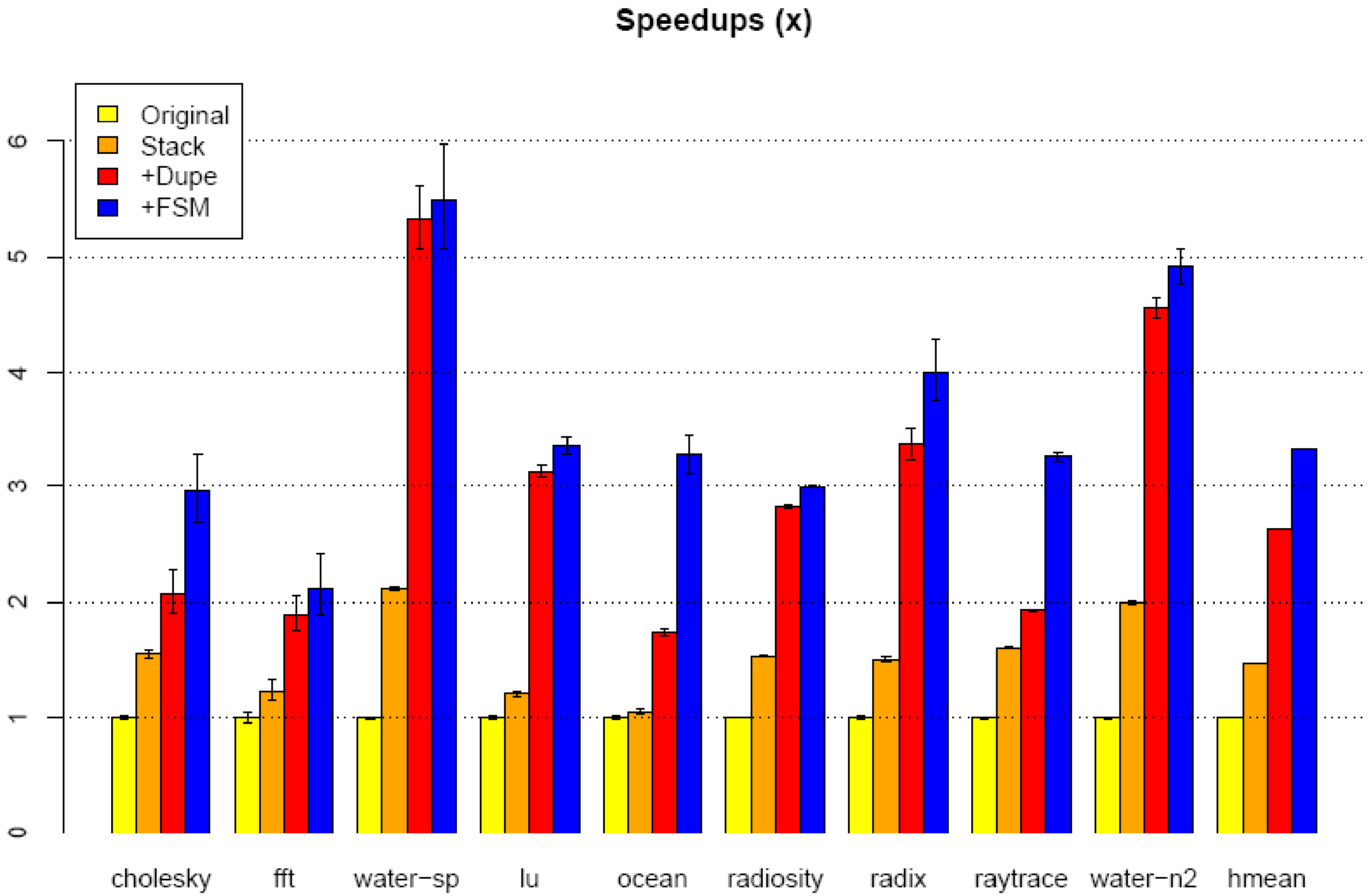
Fraction of References Filtered Out



Incremental Filtering Rate



Thread Checker Speedups



No Data Races Missed

Application	Races found	Filter Rate (%)	Original Overhead (x)	Overhead w/ Filter (x)
Cholesky	1/1	99	239	72
FFT	0/0	92	90	37
LU	1/1	97	428	125
Ocean	1/1	97	90	26
Radiosity	5/5	99	485	162
Radix	2/2	99	222	52
Raytrace	2/2	98	172	52
Water-sp	0/0	99	183	30
Water-n2	0/0	99	189	38
Average		97	233	66

Conclusion

- Three filters developed:
  - Stack
  - Duplicate
  - FSM
- Filter 97% of accesses without losing races
  - Could be used with many data-race detectors
- Avg. slowdown reduced from 233x to 66x
  - Much more work to be done

Accurate and Efficient Filtering for the Intel Thread Checker Race Detector

Paul Sack, Brian E. Bliss*, Zhiqiang Ma*,
Paul Petersen*, and Josep Torrellas
*University of Illinois and Intel Corporation**

October 21, 2006