Detecting and Tolerating Asymmetric Races

[1]Paruj Ratanaworabhan, [2]Martin Burtscher, [3]Darko Kirovski
[3]Benjamin Zorn, [4]Rahul Nagpal, and [5]Karthik Pattabiraman

[1]Cornell University
[2]The University of Texas at Austin
[3]Microsoft Research Redmond
[4]Indian Institute of Science
[5]University of Illinois at Urbana-Champaign

Overview

- Introduction of multicore processors made parallel programming ubiquitous
- Parallel programming is hard
  - Suffers from all the problems of sequential programming
  - Introduces additional sources of errors
    - E.g., deadlock, atomicity violation, and data races

Scope of this work

- Data races
  - Asymmetric races
- Large code base of parallel applications
  - Lock-based programs
  - Written mostly in C/C++
  - Use add-on libraries for threading and synchronization
- *ToleRace*: detects and tolerates races at runtime

Talk outline

- Overview and scope
- Asymmetric races
- The oracle ToleRace
- Pin-ToleRace
- Evaluation of Pin-ToleRace
- Ideal software ToleRace
- Summary

Asymmetric races

- One thread correctly protects a shared variable
- Another thread accesses the same variable with improper synchronization

```
Thread 1:
CSEnter(mutex_A)
if (gScript == NULL)
  baseScript = default;
else
  baseScript = gScript;
CSExit(mutex_A)
compile(baseScript)

Thread 2:
gScript = NULL
```

Why focus on asymmetric races

- Prevalent in software development projects
  - Direct experience from Microsoft developers
  - Possible reasons:
    - Correct local reasoning but lock convention broken
    - Assumptions in legacy codes invalidated

- Symmetric races are often benign

```
// K and flag are declared volatile
Thread 1:              Thread 2:
K = x;                 while (flag != true);
flag = true;           y = K;
```

Characterizing asymmetric races

T_1 = safe thread taking proper locks t_2 = unsafe thread improperly synchronized

$T'_1\ T''_1\ t_2$	*race*
R+ R+WX* wx*	true (non-repeatable read)
WX* wx* WX*	false = wx* WX* WX*

no race = T_1 and t_2 operations are serializable

Upper case for safe thread

Lower case for unsafe thread

Read (r, R) Write (w, W) Don't care (x, X)

Read-dependent write (rw, RW)

+ denotes one or more of the preceding operation

* denotes zero or more of the preceding operation

Characterizing asymmetric races

Possible interaction sequences: R+(r+), WX*(wx*), and R+WX*(r+wx*)

T_1 = safe thread taking proper locks t_2 = unsafe thread improperly synchronized

T'_1	t_2	T''_1	race		T'_1	t_2	T''_1	race		T'_1	t_2	T''_1	race	
R+	r+	R+	false		R+	wx*	R+	true	I	R+	r+wx*	R+	true	IV
R+	r+	WX*	false		R+	wx*	WX*	true	III	R+	r+wx*	WX*	true	IV
R+	r+	R+WX*	false		R+	wx*	R+WX*	true	I	R+	r+wx*	R+WX*	true	IV
WX*	r+	R+	false		WX*	wx*	R+	true	I	WX*	r+wx*	R+	true	IV
WX*	r+	WX*	true	II	WX*	wx*	WX*	false		WX*	r+wx*	WX*	true	IV
WX*	r+	R+WX*	true	II	WX*	wx*	R+WX*	true	I	WX*	r+wx*	R+WX*	true	IV
R+WX*	r+	R+	false		R+WX*	wx*	R+	true	I	R+WX*	r+wx*	R+	true	IV
R+WX*	r+	WX*	true	II	R+WX*	wx*	WX*	true	III	R+WX*	r+wx*	WX*	true	IV
R+WX*	r+	R+WX*	true	II	R+WX*	wx*	R+WX*	true	I	R+WX*	r+wx*	R+WX*	true	IV

no race = T_1 and t_2 operations are serializable

- Race case:

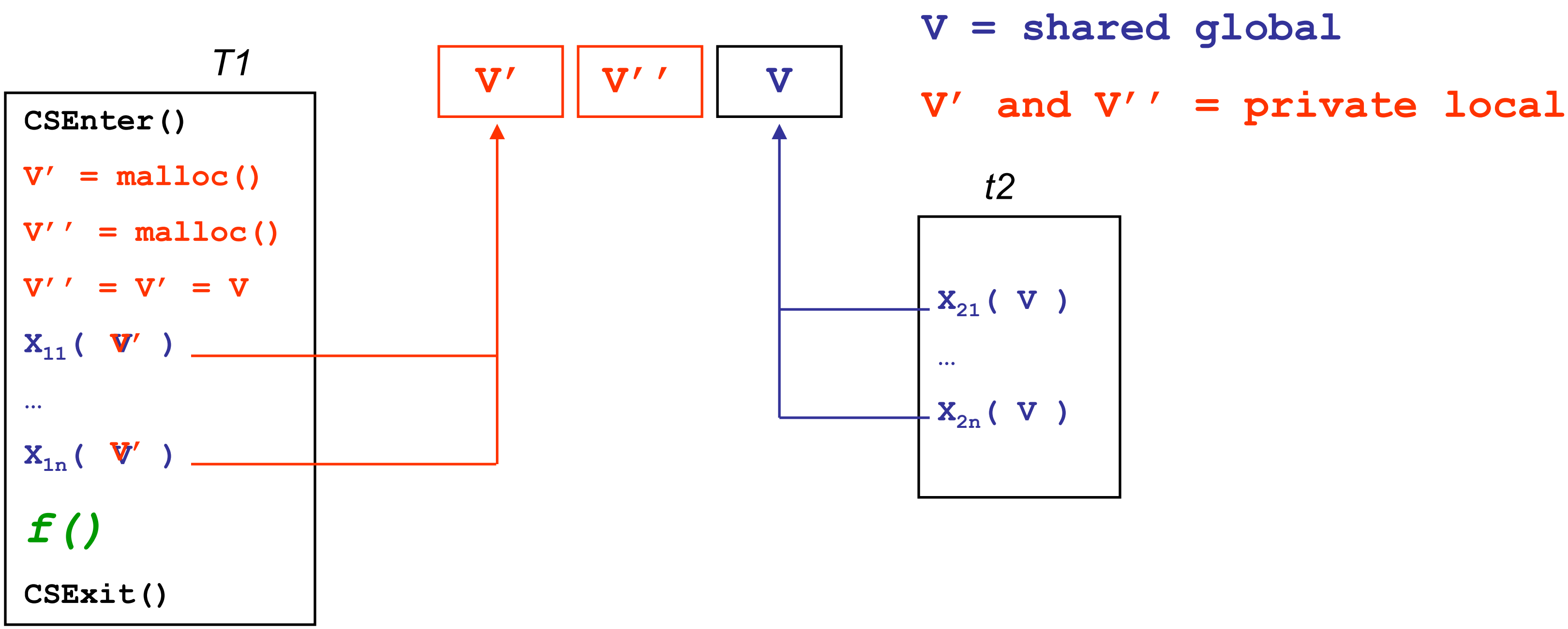
  I: XwR II: WrW III: RwW IV: XrwX

- Any race conditions among K>2 threads can always be reduced to one of the four race cases

Talk outline

- Overview and scope
- Asymmetric races
- The oracle ToleRace
- Pin-ToleRace
- Evaluation of Pin-ToleRace
- Ideal software ToleRace
- Summary

The oracle ToleRace mechanics



- Upon acquiring a lock protecting V, T1 creates two private copies of V: V' and V''
- T1 operates on V'; t2 on V; V'' is a clean copy
- Before unlocking, execute the resolution function *f*
  - Detecting and/or tolerating the race at this point

ToleRace resolution function

- Given a shared variable:

  V = global operable; V’ = local operable; V’’ = local clean

- Tolerate races by enforcing serial execution

- Race cases:

I: XwR	II: WrW	III: RwW	IV_A: XrwX / RrwR	IV_B: WrwX	IV_C: RrwW
V != V’’	**V == V’’**	**V != V’’**	**V != V’’**	**V != V’’**	**V != V’’**
SE: XRw	**SE: rWW**	**SE: RXWw**	**SE: RRrw**	**SE: rwWX**	**N/A**
***F()* = V**	***F()* = V’**	***F()* = V**	***F()* = V**	***F()* = V’**	***custom F()***
T_1t_2	**t_2T_1**	**T_1t_2**	**T_1t_2**	**t_2T_1**	**N/A**
Detect & Tolerate	**Tolerate**	**Detect & Tolerate**	**Detect & Tolerate**	**Detect & Tolerate**	**Detect**

Analogous to transactional memory, …

Comparison with Transactional Memory

- Uses lazy versioning and lazy conflict detection
- Never aborts or rolls back
- Does not need contention management
- Can handle I/O and overlapped critical sections

Talk outline

- Overview and scope
- Asymmetric races
- The oracle ToleRace
- Pin-ToleRace
- Evaluation of Pin-ToleRace
- Ideal software ToleRace
- Summary

Pin-ToleRace

- Implemented software ToleRace on top of Pin, a dynamic instrumentation tool from Intel
- Work directly on executables
- Motivation for software ToleRace:
  - Can be deployed immediately
  - Gauges the worst case overhead by performing all analyses and decisions at runtime

Pin-ToleRace specific details

- x86/Linux platform
- Parallel pthread-based programs
- pthread_mutex_lock/unlock pair defines a critical section

Oracle ToleRace versus Pin-ToleRace

- Oracle ToleRace
  - Protects shared variables
  - All protected variables known
  - Atomic copy
  - Not realizable
- Pin-ToleRace
  - Protects shared memory
  - All protected locations determined on-the-fly
  - Non-atomic copy
  - Implementable

Tolerate races with Pin-ToleRace

- Pin-ToleRace knows all the shared accesses in the safe thread, but cannot distinguish between intervening **rw** and **w** sequences from other threads
- Comparison of oracle ToleRace with Pin-ToleRace

race type		Tolerable	
		Oracle ToleRace	Pin-ToleRace
I	XwR	**true**	**true**
II	WrW	**true**	**true**
III	RwW	**true**	**false**
IV_A	RrwR	**true**	**true**
IV_B	WrwX	**true**	**true**
IV_C	RrwW	**false**	**false**

Pin-ToleRace evaluation

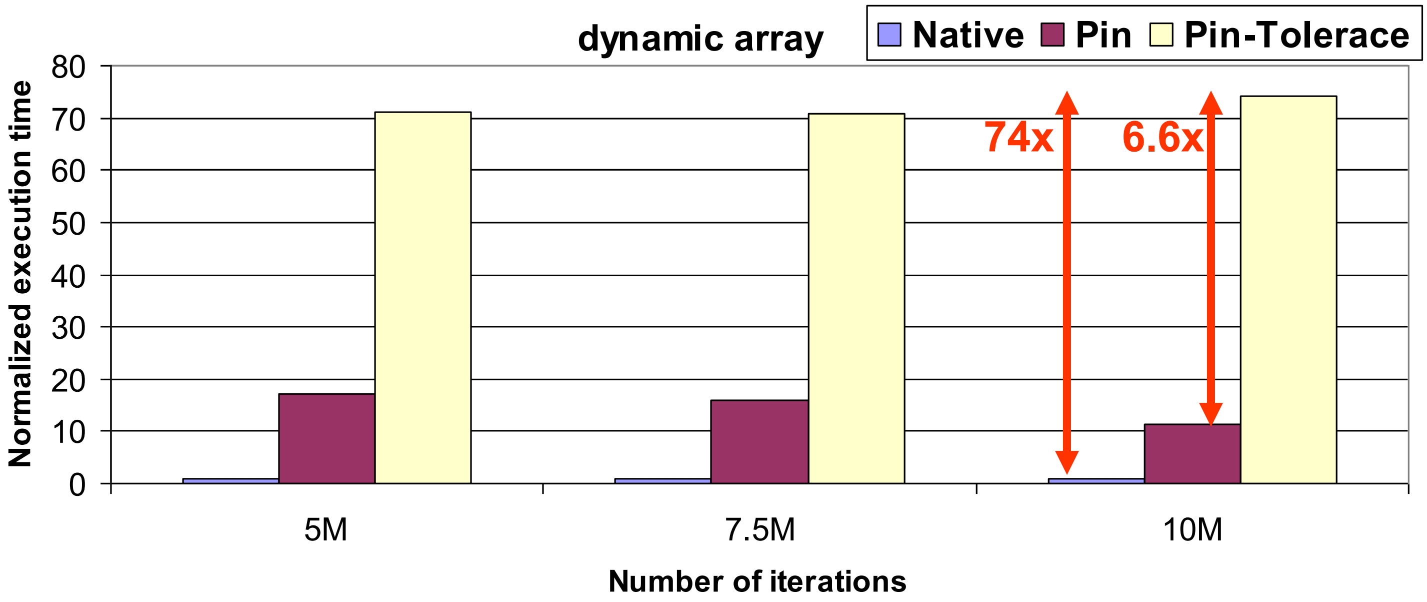
- Microbenchmark stress tests
- Real applications

Benchmarks

- 3 microbenchmarks for stress tests
  - Scalar, static array, and dynamic array
- 13 real applications
  - SPLASH2: four kernels and four applications
  - PARSEC: one kernel and four applications
- All benchmarks compiled and run on Intel 32-bit system with 4-core 2.8 GHz P4-Xeon

Stress tests

- Demonstrate race toleration with microbenchmarks
  - Safe thread: increments shared counters each iteration
  - Unsafe threads: impart random writes to shared counters
- Overhead is very high
  - Almost always executing inside critical sections



Critical section characteristics

	unique	nested CS	dynamic number of instrs per CS (user)	% dynamic instrs in CS
cholesky	14	no	29	< 0.1%
fft	10	no	17	< 0.01%
lu	7	no	17	< 0.01%
radix	9	no	17	< 0.01%
barnes	10	no	94	0.18%
ocean	26	no	17	< 0.01%
radiosity	36	yes	18	0.11%
water-spatial	16	no	13	< 0.01%
dedup	7	yes	600	0.42%
facesim	5	yes	46	< 0.01%
ferret	4	yes	690	1.59%
fluidanimate	11	no	13	0.40%
x264	2	no	11	< 0.01%

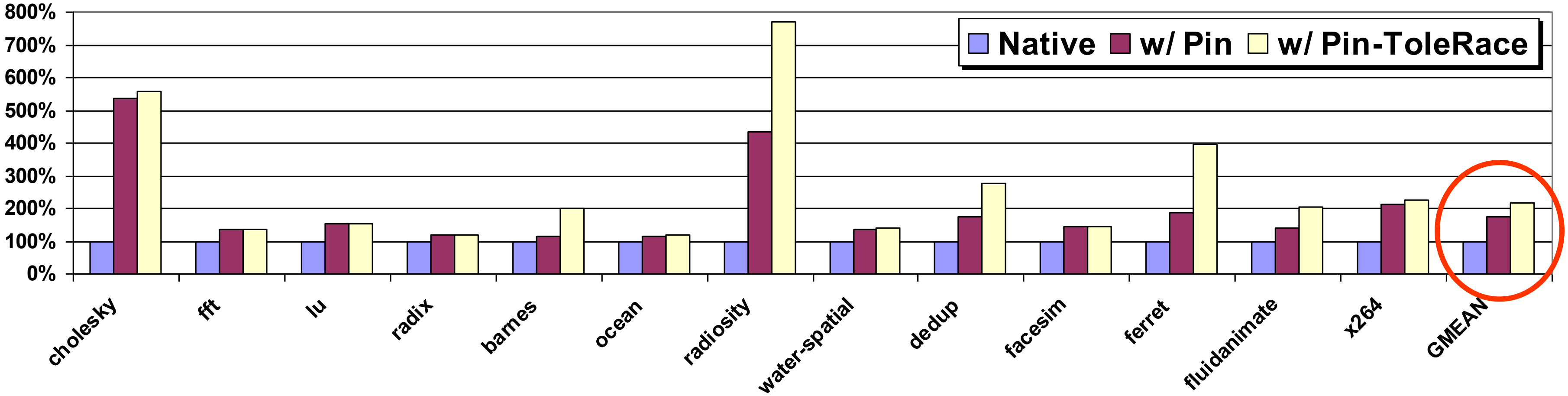
- Small number of unique critical sections
- Infrequently executing inside critical sections

Critical section characteristics

- The table shows unique accesses to possibly shared locations per critical section
- This number is less than five except for barnes, dedup, facesim, and ferret

	unique accesses	
	AVG	STD
cholesky	4.78	0.38
fft	1.37	0.04
lu	2.99	0.01
radix	2.82	0.19
barnes	19.13	0.03
ocean	3.00	0.00
radiosity	4.92	0.23
water-spatial	2.62	0.01
dedup	80.87	3.52
facesim	7.70	1.14
ferret	72.89	33.83
fluidanimate	5.00	0.00
x264	2.16	0.02

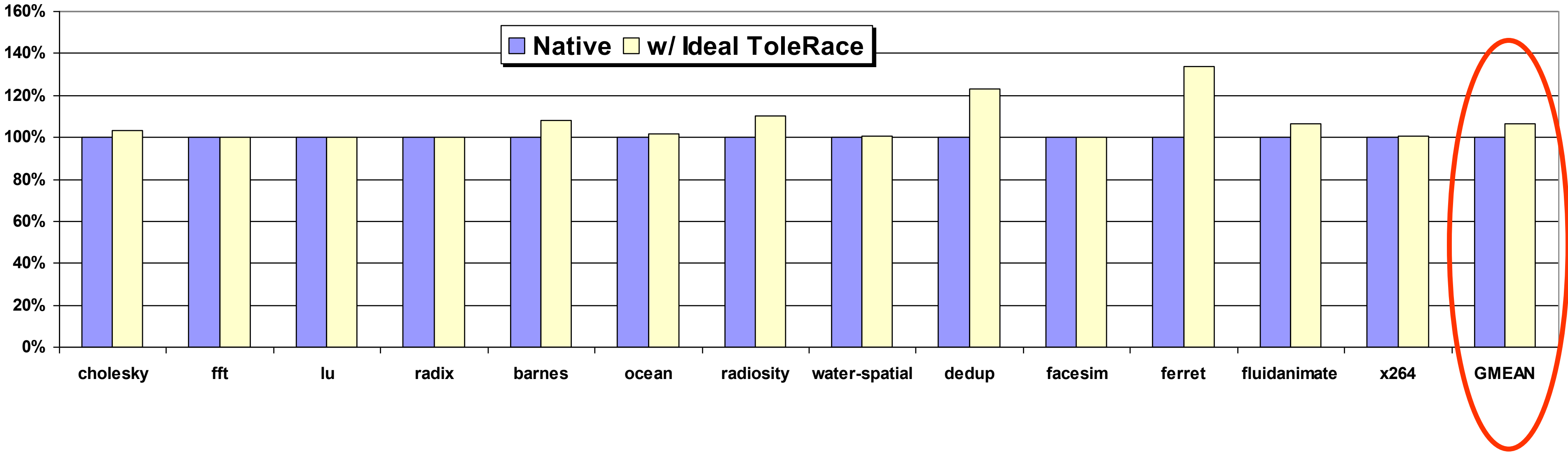
Pin-ToleRace performance



Normalized execution time of Pin-ToleRace

- On average, about 2X and 24% slowdown compared to the native and Pin run, respectively
- Approximate upper bound on overhead

Ideal software ToleRace performance



Normalized execution time of ideal software ToleRace

- On average, only 7% slowdown
- Most applications run with less than 1% overhead
- Approximate lower bound on overhead

Summary

- Asymmetric races are an important class of parallel programming errors
- We presented ToleRace, a theoretical framework for detecting and *tolerating* asymmetric races
- We showed that an implementable software ToleRace system based on Pin has a 2X overhead
- Aim to further improve ToleRace by
  – Providing a stronger isolation guarantee
  – Lowering the software ToleRace overhead

Backup Slides

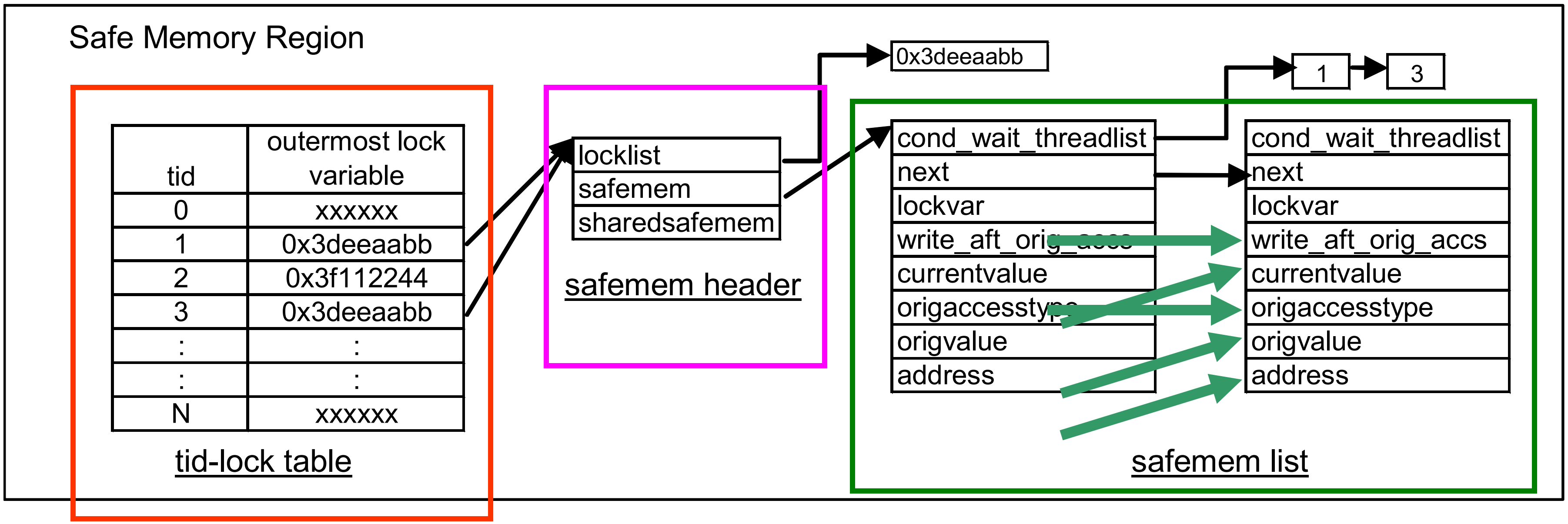
The oracle ToleRace

- A *theoretical* framework for handling asymmetric races in lock-based parallel programs
  - Creates local copies of shared variables upon CSEnter()
  - Detects changes to shared data at critical CSExit()
  - Propagates the appropriate copy to hide races
- Dynamically detects the race and also tolerates it whenever possible
- Incurs overhead only at critical section execution

Pin-ToleRace general framework

- Defines safe memory as the region that holds local copies of shared memory locations
- Once in a critical section, instruments each executed instruction touching shared locations
- Searches the safe memory for shared locations
  - If found: accesses are redirected to the safe memory
  - If not: create a new node in the safe memory and redirect accesses

The safe memory region



- Contains three main data structures:
  - Safemem list
  - Tid-lock table
  - Safemem header