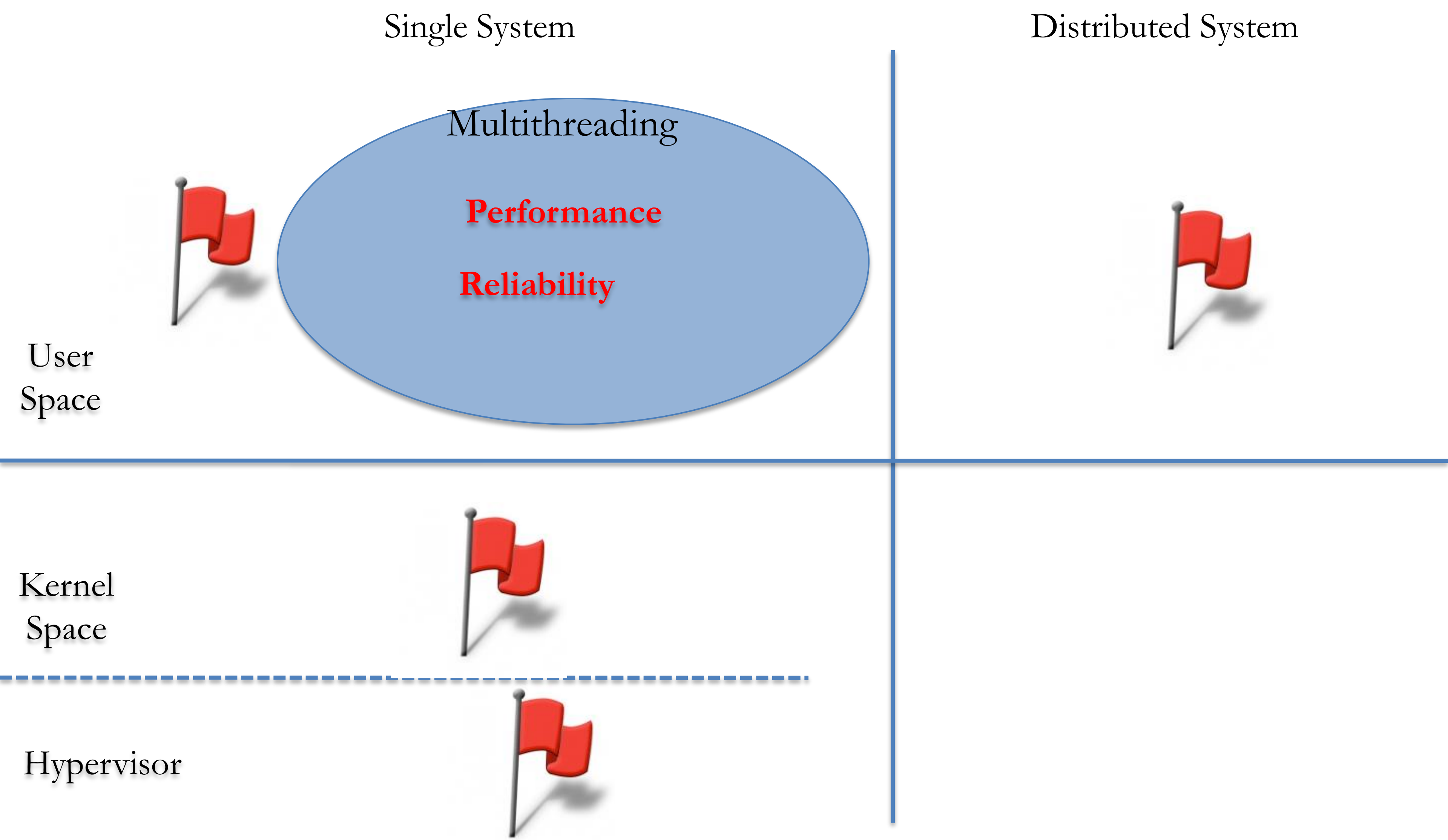
Systems Research Group

Tongping Liu

http://www.cs.utsa.edu/~tongpingliu/

Broad Research Field

MY MISSION: help programmers design correct and efficient software systems



Performance Improvement for Parallel Applications

Tongping Liu

http://people.cs.umass.edu/~tonyliu

Parallelism is Important

- Multicore is the standard
  - smart phones, tablets
  - laptops, workstations
  - supercomputers, data centers

Multicore drives parallel computing

Parallel Computing is Challenging

- Efficiency Problem
  - Algorithm, data structure
  - Type and distribution of workload (parallelizable percentage, task granularity, load balance, thread model)
  - Hardware effect
- Reliability Problem
  - Input dependent
  - Timing dependent

Parallel Computing is Challenging

- Efficiency Problem
  - Algorithm, data structure
  - Type and distribution of workload (parallelizable percentage, task granularity, load balance, thread model, locality)
  - Hardware effect — False sharing on cache lines: SHERIFF, PREDATOR
- Reliability Problem
  - Input dependent — Memory error: DOUBLETAKE
  - Timing dependent — Deterministic Multithreading: DTHREADS

Research Focus: Parallel Computing

Performance

SHERIFF: [Liu, OOPSLA'11]

Detecting and Tolerating False Sharing

PREDATOR: [Liu, PPOPP'14]

Predictive False Sharing Detection

Reliability

DTHREADS: [Liu, SOSP'11]

Efficient Deterministic Multithreading

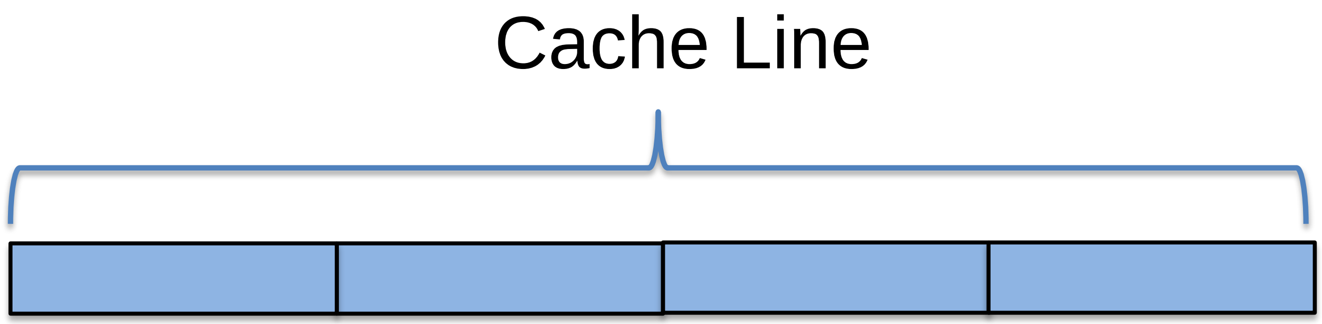
DOUBLETAKE: [Liu, Submission]

Evidence-Triggered Dynamic Analysis

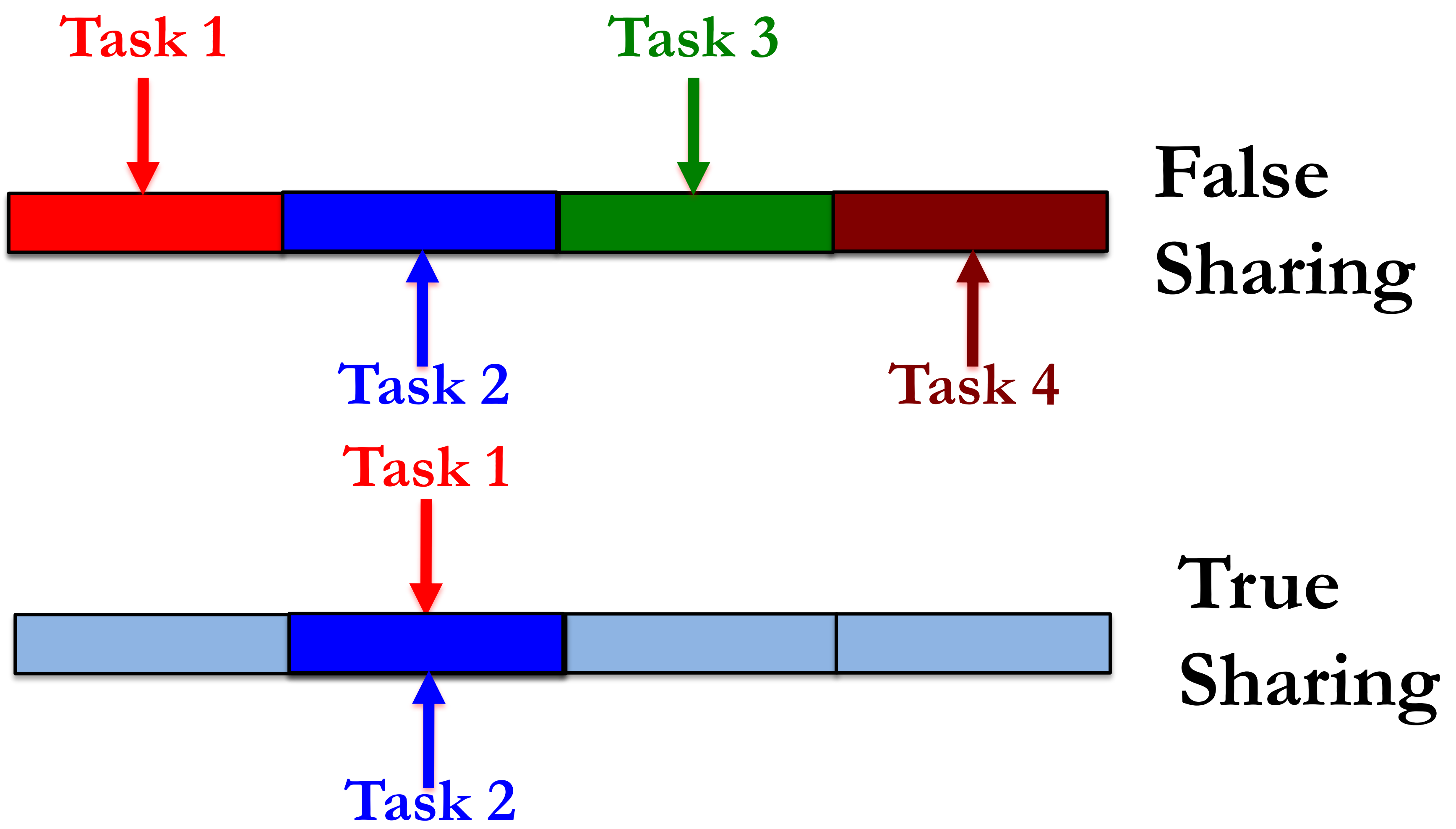
Outline

- False Sharing: Background & Motivation
- Correctly and Precisely Detect False Sharing
- Automatically Eliminate False Sharing
- Other Contributions
- Future Work

False Sharing vs. True Sharing



False Sharing vs. True Sharing



False Sharing can dramatically degrade performance

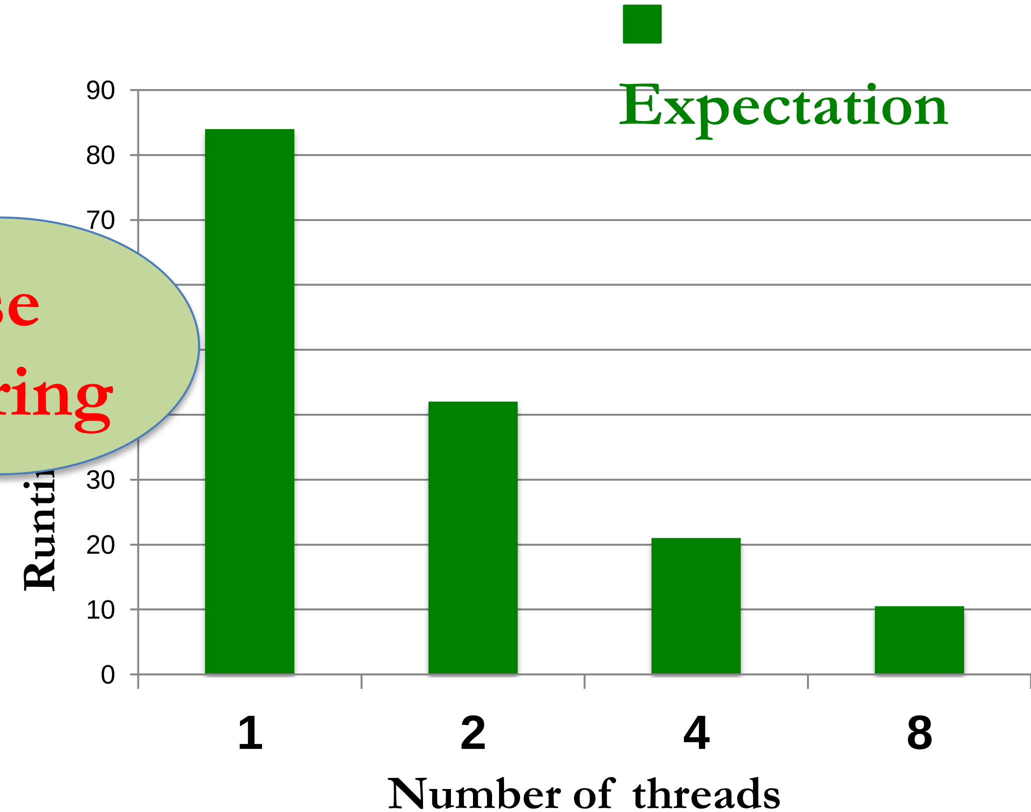
Parallelism: Awesome Expectation

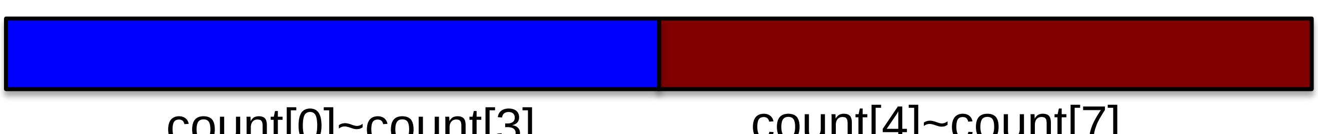
Parallel Program

```
int count[8];
int W;
void increment(int S)
{
   for(in=S; in<S+W; i
     for(j=0; j<1M; j+
       count[in]++;
}

int main(int THREADS) {
 W=8/THREADS;
 for(i=0; i<8; i+=W)
  spawn(increment,i);
}
```

False sharing





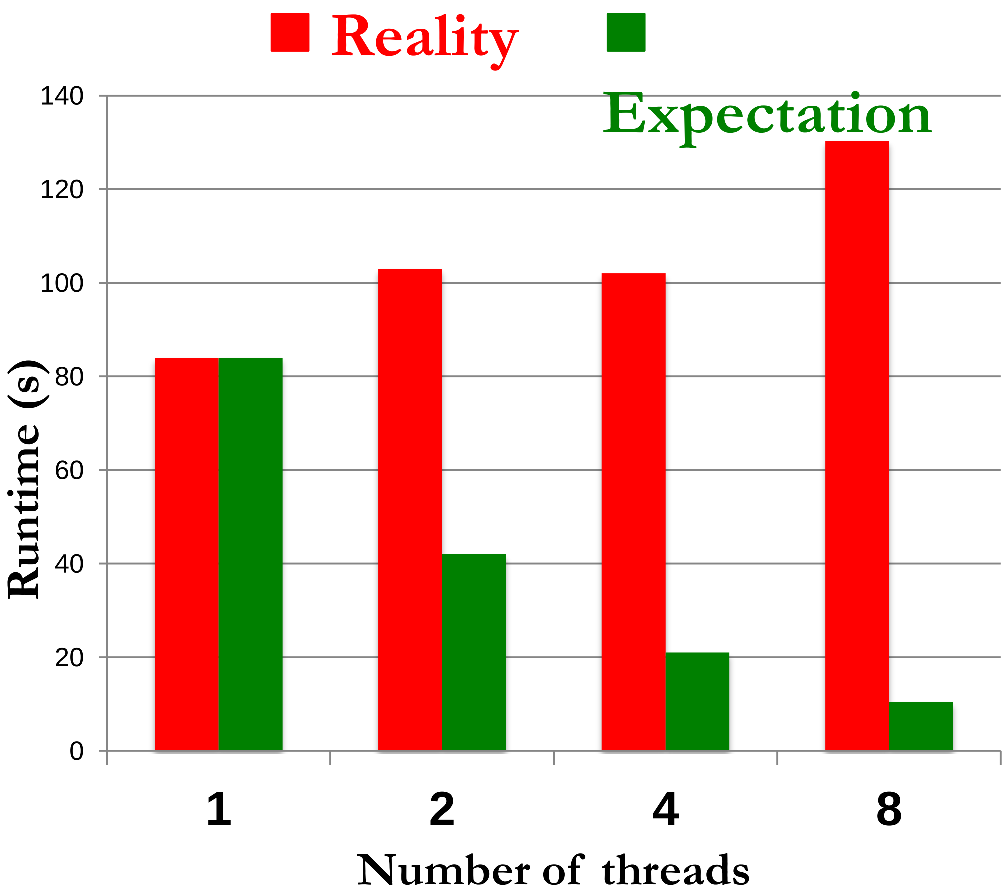
THREADS = 2

Parallelism: Awful Reality

Parallel Program

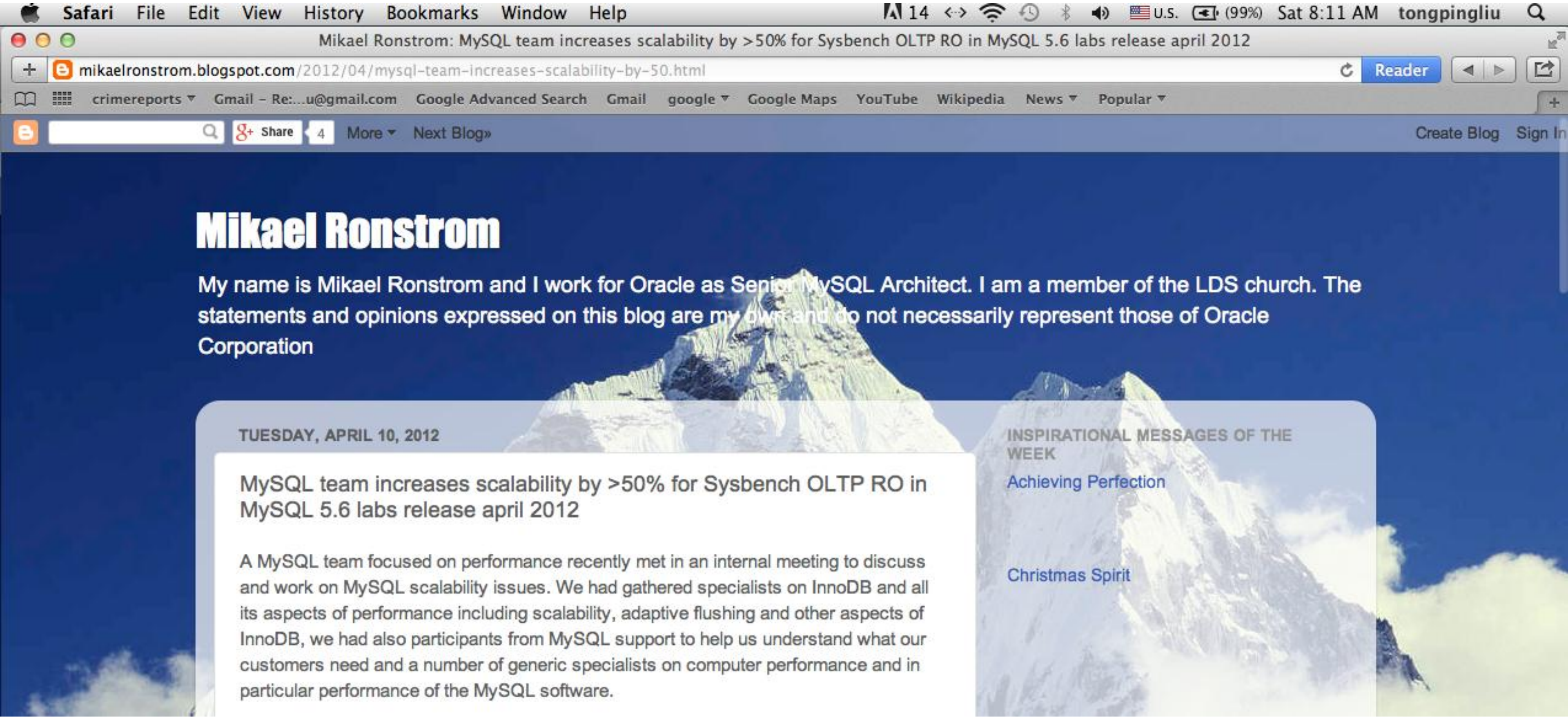
```
int count[8];
int W;
void increment(int S)
{
  for(in=S; in<S+W; in++)
    for(j=0; j<1M; j++)
      count[in]++;
}

int main(int THREADS) {
 W=8/THREADS;
 for(i=0; i<8; i+=W)
  spawn(increment,i);
}
```



False sharing slows the program by 13X

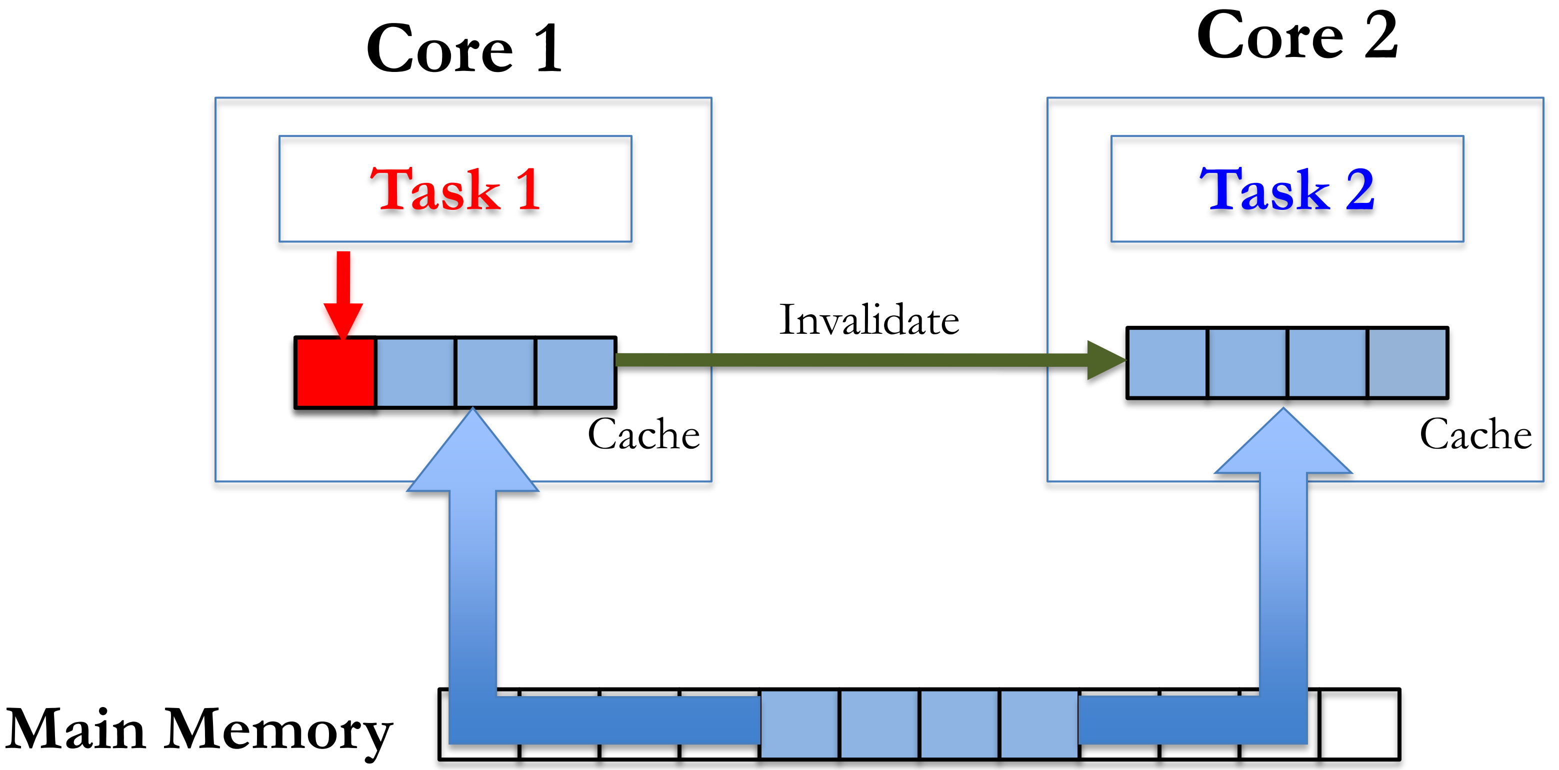
False Sharing in Real Applications



False sharing slows MySQL by 50%

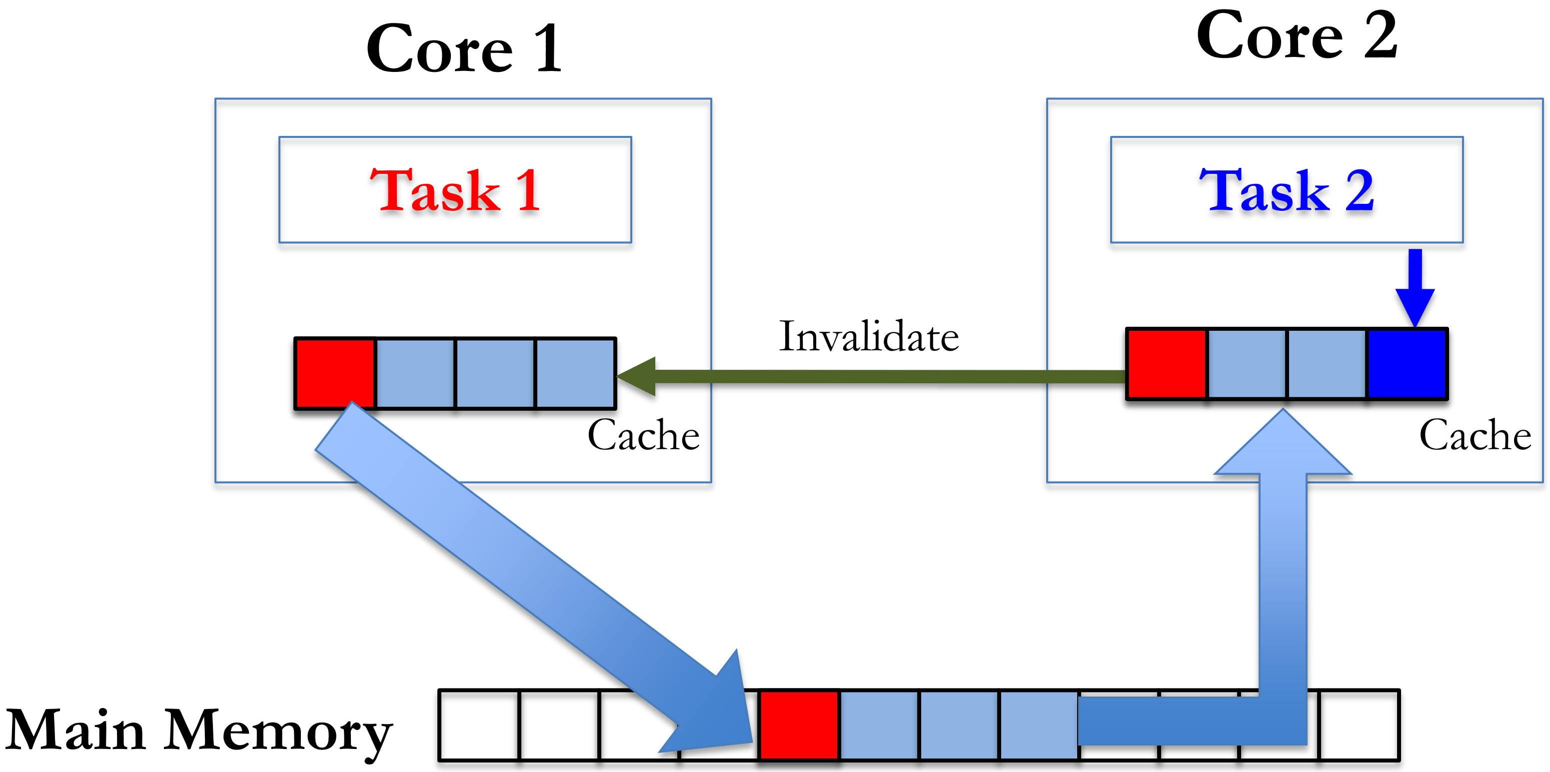
Resource Contention at Cache Line Level

False Sharing Causes Performance Problems



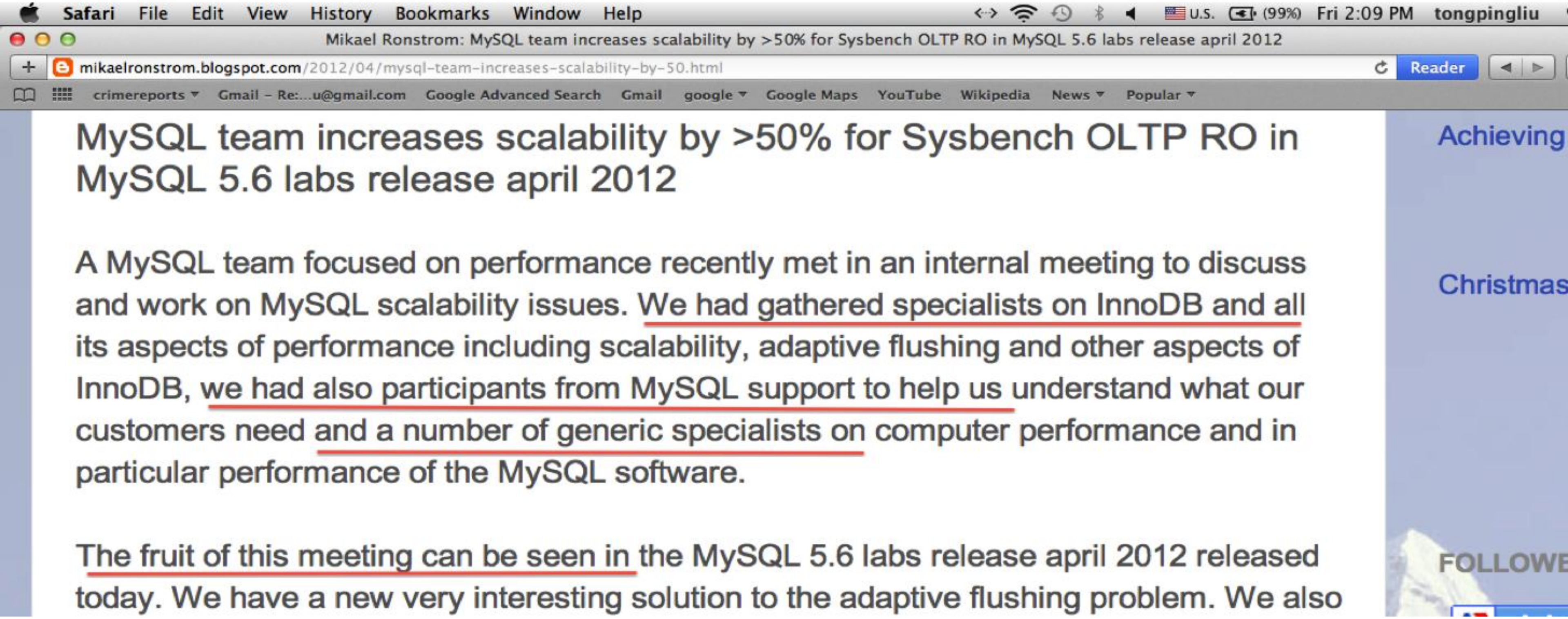
Cache line: basic unit of data transfer

False Sharing Causes Performance Problems



Interleaved accesses cause cache invalidations

False Sharing is Hard to Diagnose



Multiple experts worked together to diagnose MySQL scalability issue (1.5M LOC)

PREDATOR: Predictive False Sharing Detection

Tongping Liu, Chen Tian, Ziang Hu, Emery Berger

Interested by Many Companies

"Here IBM has this power platform with different settings from X86. **I'm thinking about techniques that can detect false sharing on Power, and your solution is quite relevant on this aspect.**"

- IBM, Intel, Huawei, SAS, Mathworks

Related Work

- S.M.Gunther et.al. WBIA 2009.

  Reports false sharing counters on physical addresses.(120X slower)

- C.Liu. Master thesis 2009.

  Reports false sharing miss ratio. (> 100X slower)

- Q.Zhao et.al. MIT. VEE2011.

  Reports cache miss ratio and cache invalidation ratio. (6X slower)

1. **False positives**
2. **Cannot pinpoint the exact cause of false sharing**

Intel Performance Tuning Utility

Basic Data Access Profiling (2010-07-12-09-33-05) Granularity Cachelines

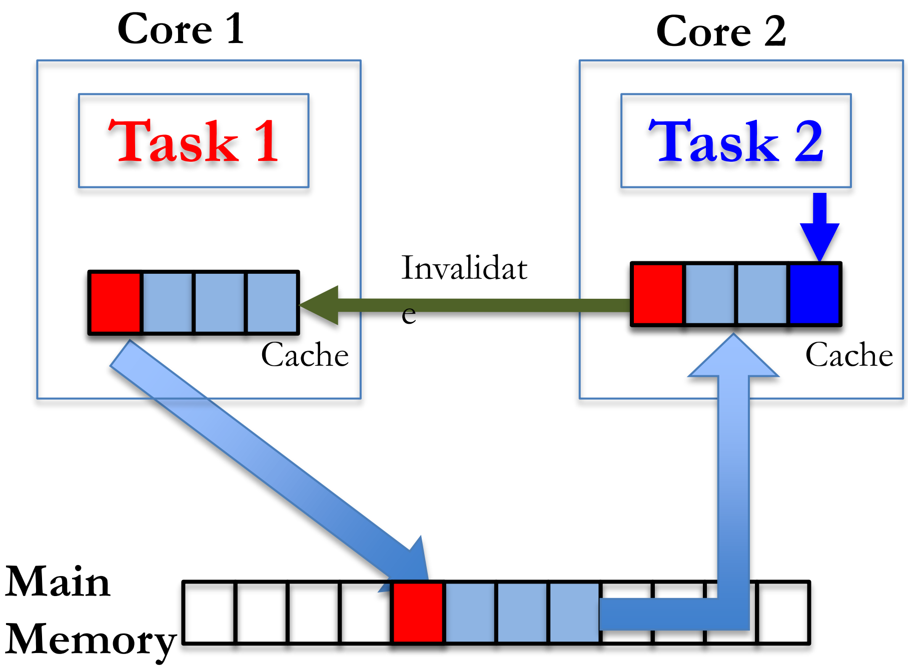
Cacheline Address / Offset / Threa...	Coll...Refs	LL...s	A...y	T...y	Contention	INST_R... refs)	M...S	MEM_LOAD_....L2_MISS	Contributors
0xef35f340	**15**	**0**	**3**	**45**	**0**	**15**	**0**	**0**	**Offsets: 3 Threa**
0xed55c340	**15**	**0**	**3**	**45**	**0**	**15**	**0**	**0**	**Offsets: 3 Threa**
0x0804f080	**12**	**0**	**4**	**99**	**2**	**3**	**9**	**0**	**Offsets: 6 Threa**
Offset:0x00(0)	4	0	10	40	0	0	4	0	Threads: 1
Thread:00004598(0011)	4	0	10	40	0	0	4	0	Functions: 1
wordcount_reduce	4	0	10	40	0	0	4	0	
Offset:0x18(24)	2	0	3	13	0	1	1	0	Threads: 1
Thread:0000459c(0014)	2	0	3	13	0	1	1	0	Functions: 1
wordcount_reduce	2	0	3	13	0	1	1	0	
Offset:0x14(20)	2	0	3	13	0	1	1	0	Threads: 1
Offset:0x0c(12)	2	0	3	13	0	1	1	0	Threads: 1
Offset:0x1c(28)	1	0	10	10	0	0	1	0	Threads: 1
Offset:0x10(16)	1	0	10	10	0	0	1	0	Threads: 1

Too many false positives

Existing Tools vs. **PREDATOR**

Existing Tools	**PREDATOR**
False positives	**No false positives**
Cannot pinpoint where are problems	**Precisely pinpoint false sharing problems**
Only detect observed false sharing	**Predict potential false sharing without occurrences**

False Sharing Causes Performance Problems



Interleaved accesses

Cache invalidations

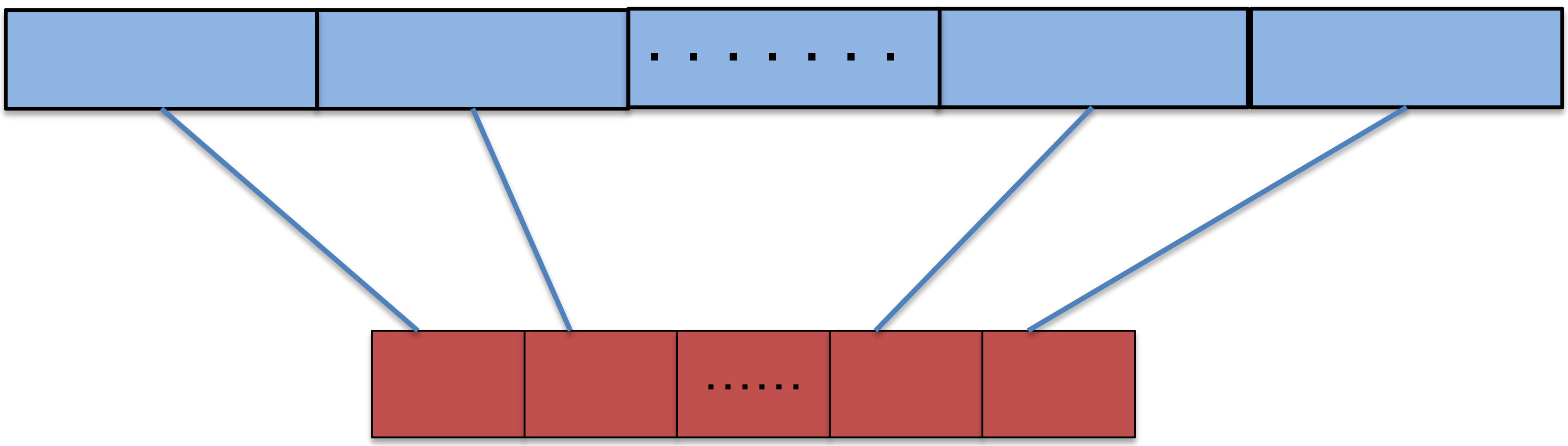
Performance problems

Detect false sharing causing performance problems

Find cache lines with many cache invalidations

Find Cache Lines with Many Invalidations

Memory: Global, Heap



Track cache invalidations on each cache line

Track Invalidations Based on Memory Accesses

Two-entries-history-table

0	0

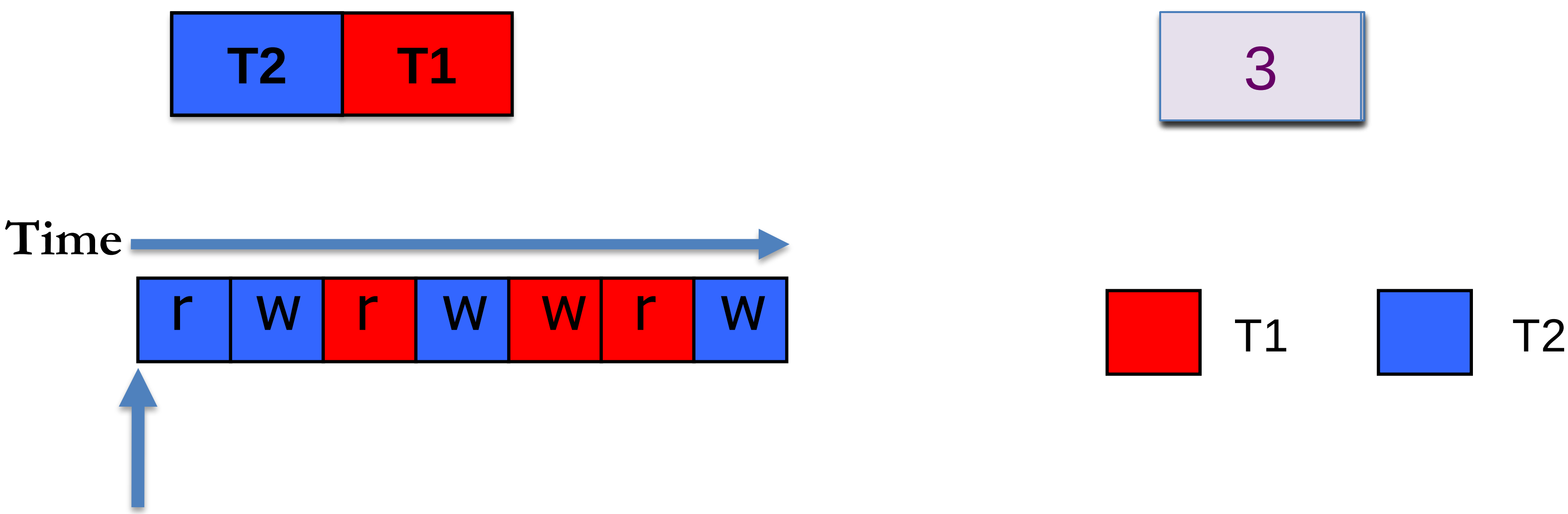
# of invalidations

0

Track Invalidations Based on Memory Accesses

Two-entries-history-table

# of invalidations



- Assumptions
  1. Each thread runs on a core with its private cache
  2. Infinite cache capacity

- **Scalable (based on tid)**
- **Portable (software-only approach)**

PREDATOR Components

Compiler Instrumentation

Instruments every memory read/write access

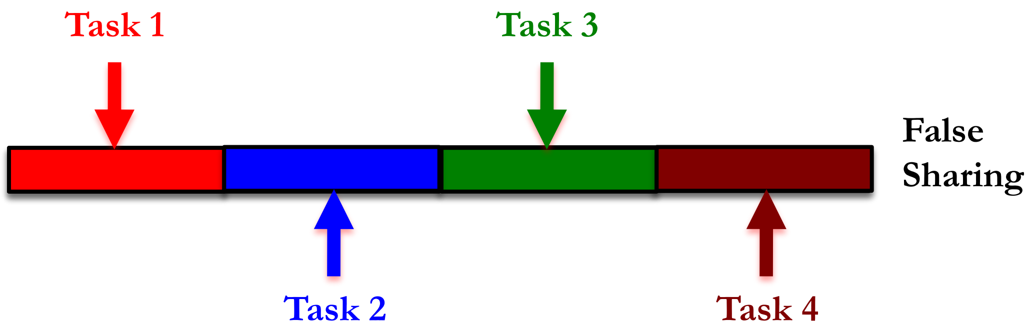
Runtime System

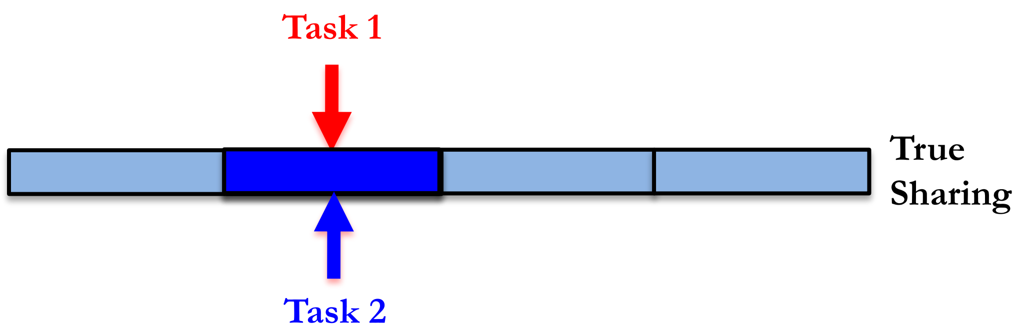
Collects memory accesses and reports false sharing

Detect Problems Correctly & Precisely

- Correctly:
  - No false alarms

Track memory accesses on each word

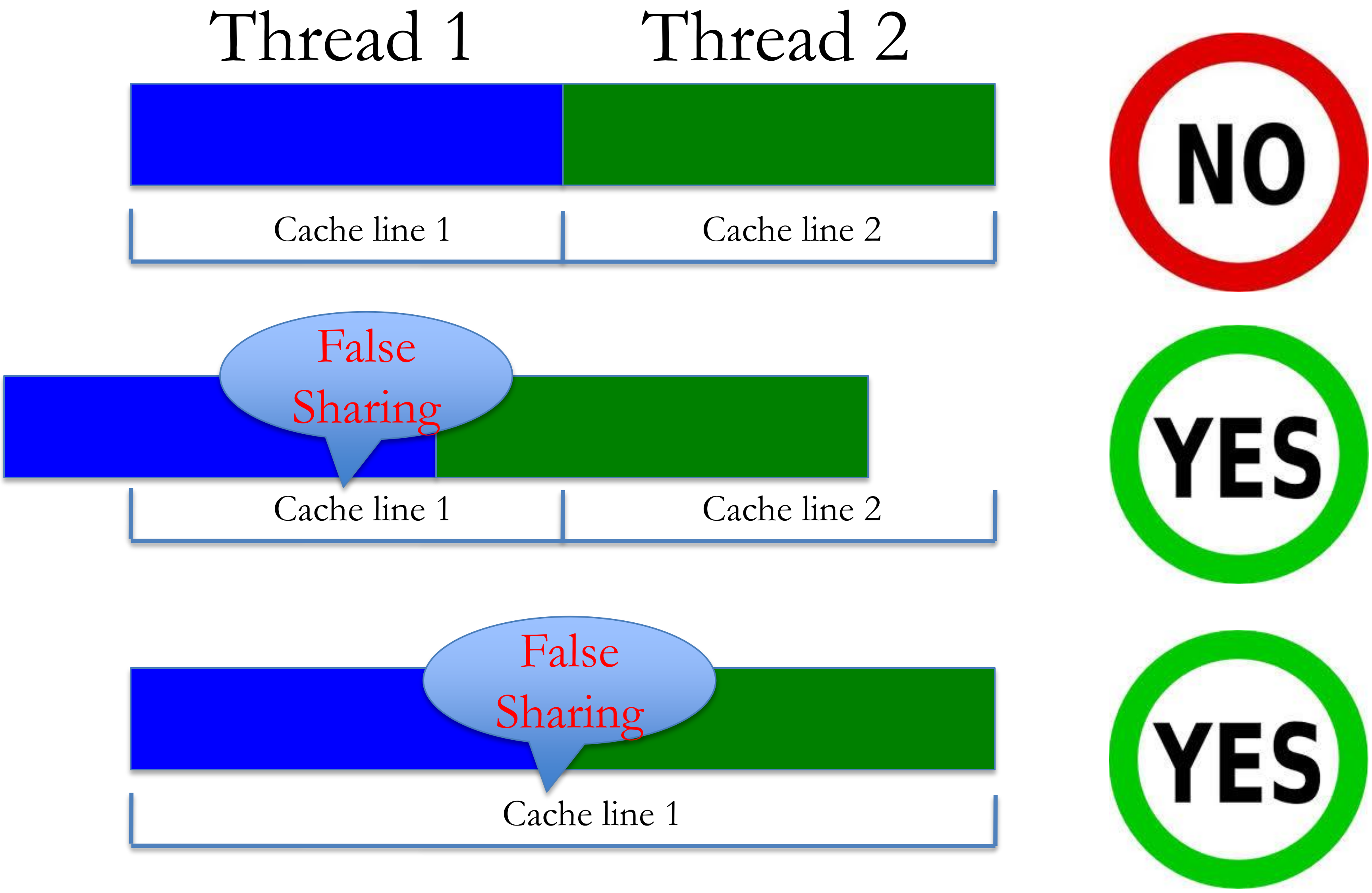




- Precisely
  - Global variables: names
  - Heap objects: calling context of memory allocation

Why do we need prediction?

Necessity of False Sharing Prediction

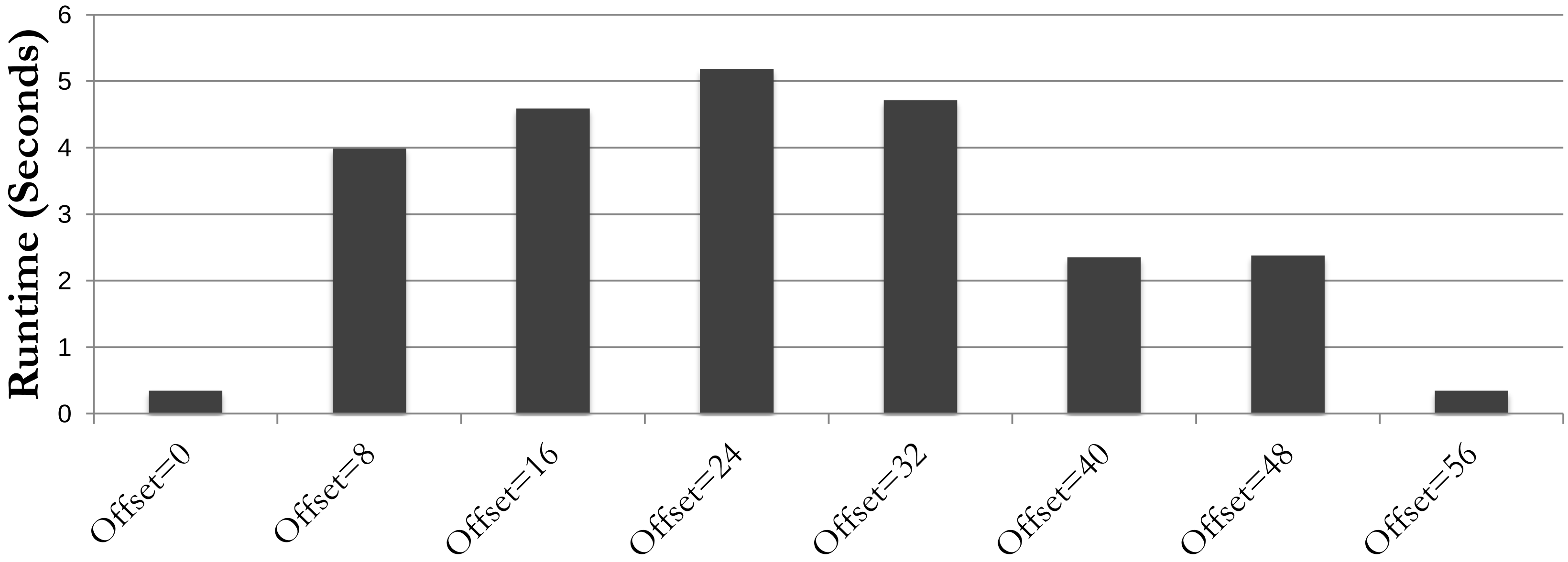


False Sharing is Sensitive to Dynamic Properties

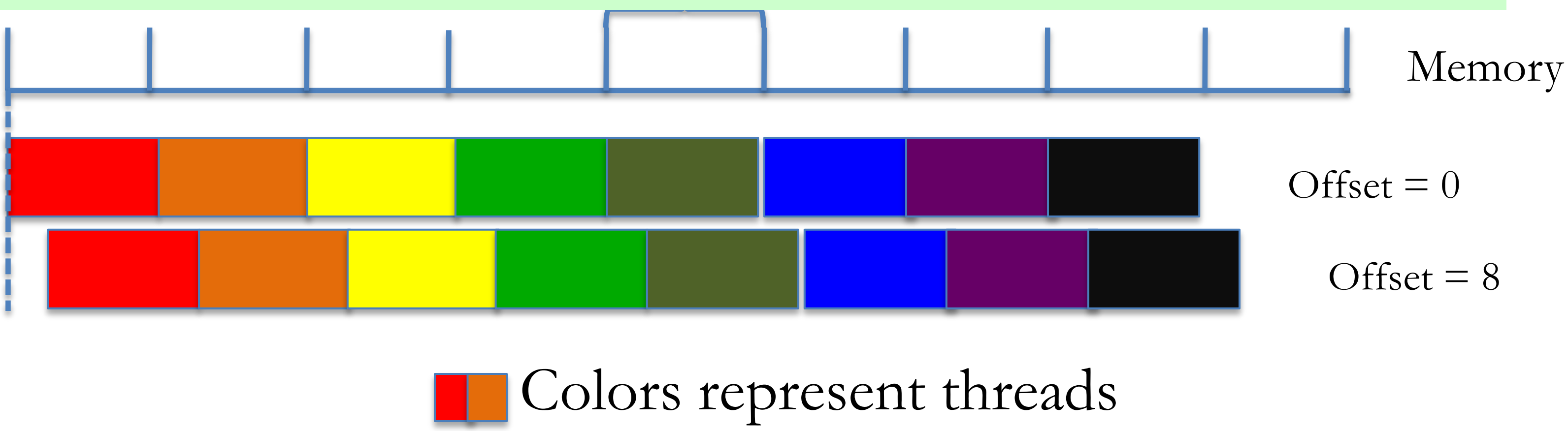
- Change of memory layout
  - ✧ 32-bit platform ← → 64-bit platform
  - ✧ Different memory allocator
  - ✧ Different compiler or optimization
  - ✧ Different allocation order by changing the code

- Change of cache line size

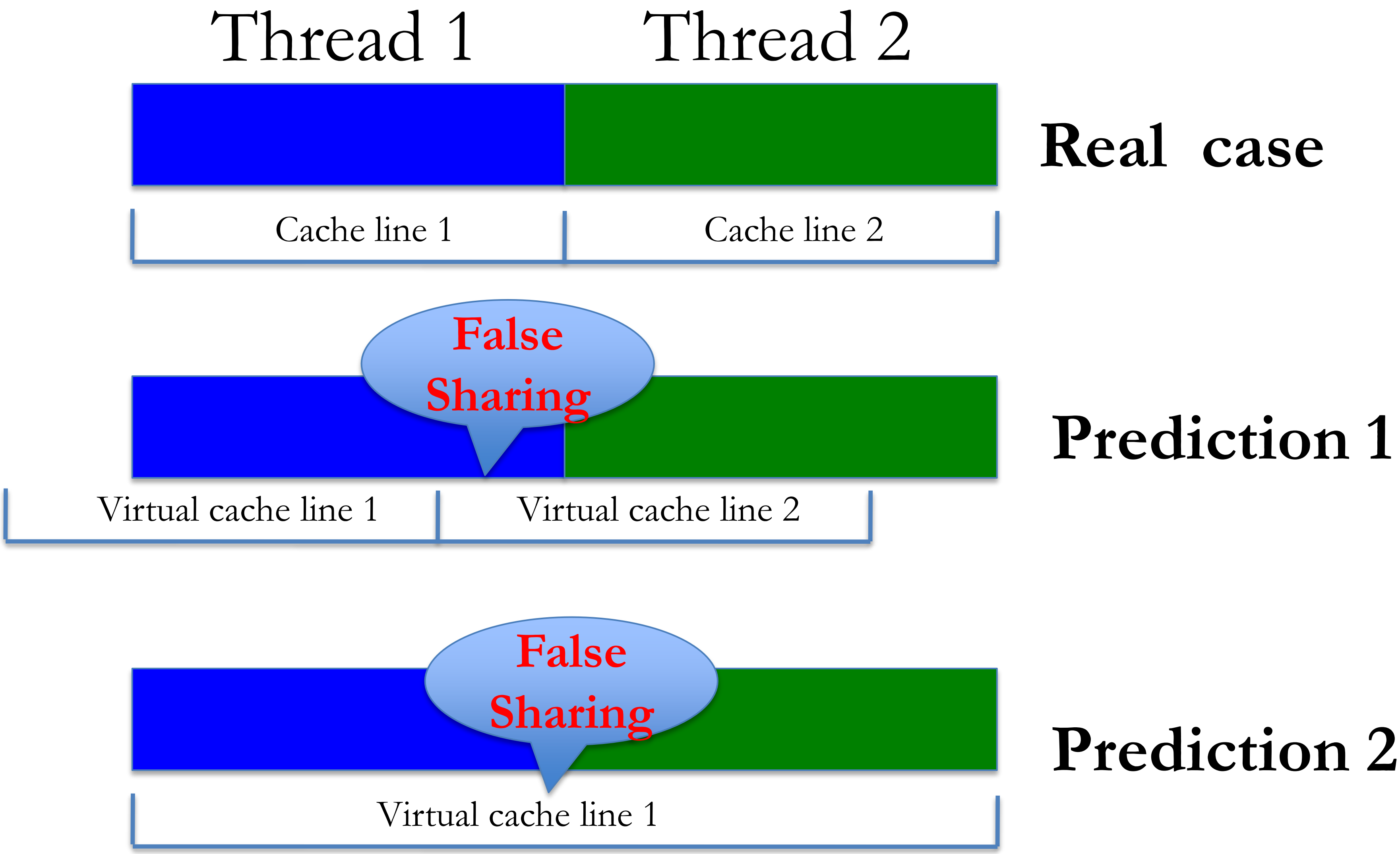
False Sharing is Sensitive to Memory Layout



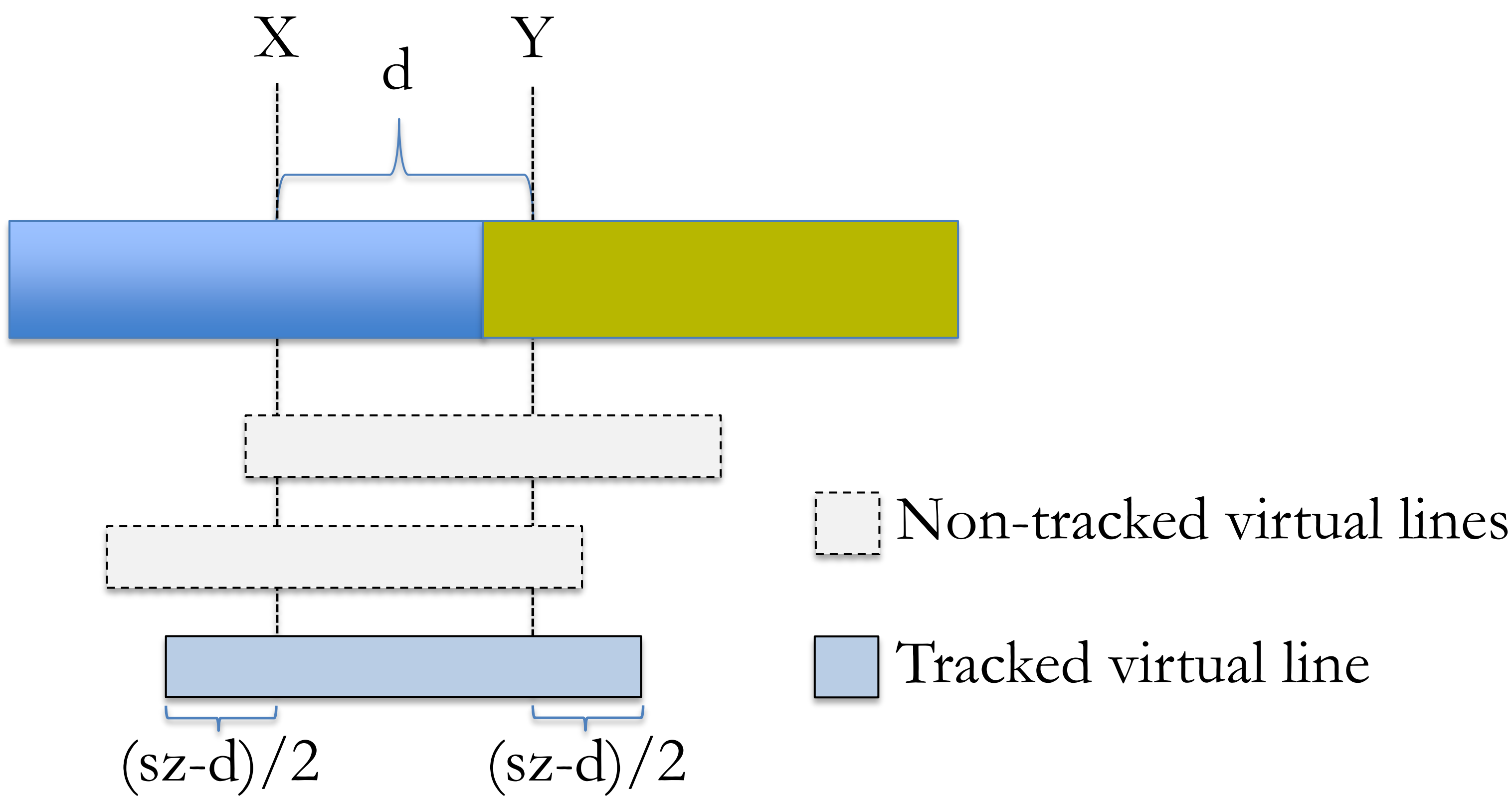
PREDATOR avoids the predicament of testing



Prediction Based on Virtual Cache Lines



Determine Virtual Line by Memory Accesses



- ✧ d < the cache line size - sz
- ✧ (X, Y) from different threads && one of them is write

Detection Results on Phoenix and PARSEC

Benchmarks	Performance Improvements (after fixes)
Histogram	**46%**
Linear_regression	**1207%**
Streamcluster-1	4.77%
Streamcluster-2	7.52%

Need prediction to detect false sharing of Linear_regression

Detection Results on Real Applications

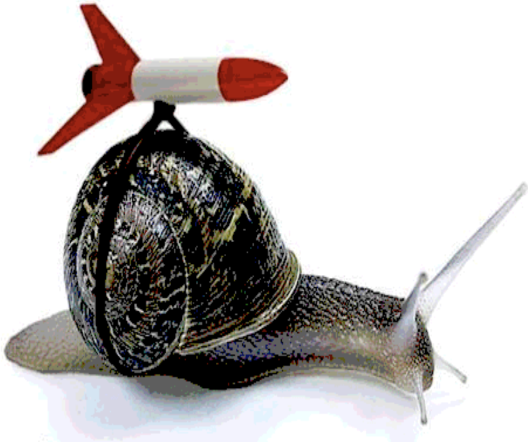
- MySQL
  - Problem: different threads update the shared bitmap simultaneously
  - Performance improves **180%** after fixes
- Boost library:
  - Problem: “there will be 16 spinlocks per cache line”
  - Performance improves about **100%**

Caveats of Fixes

- Unavailable source code
  - Infeasible to fix
- No performance improvement

Quote from the MIT's VEE2011 paper:

"We added padding between the data but the runtime actually increased because of lost cache locality."

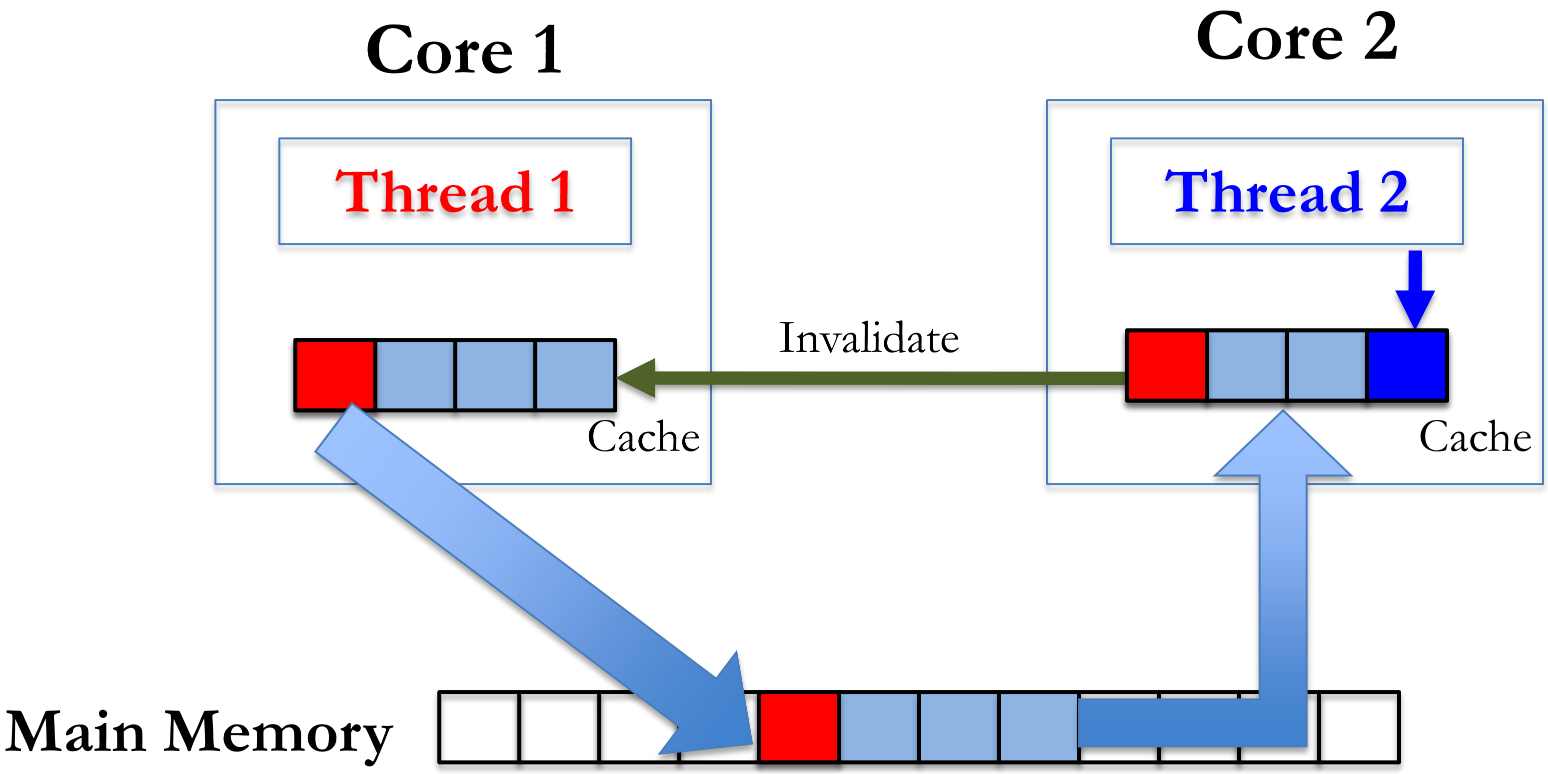


SPLASH
OOPSLA IS NOW PART OF SPLASH
PORTLAND 2011
OCTOBER 22-27

SHERIFF: Precise Detection & Automatic Mitigation of False Sharing

Tongping Liu, Emery Berger

Key Observation



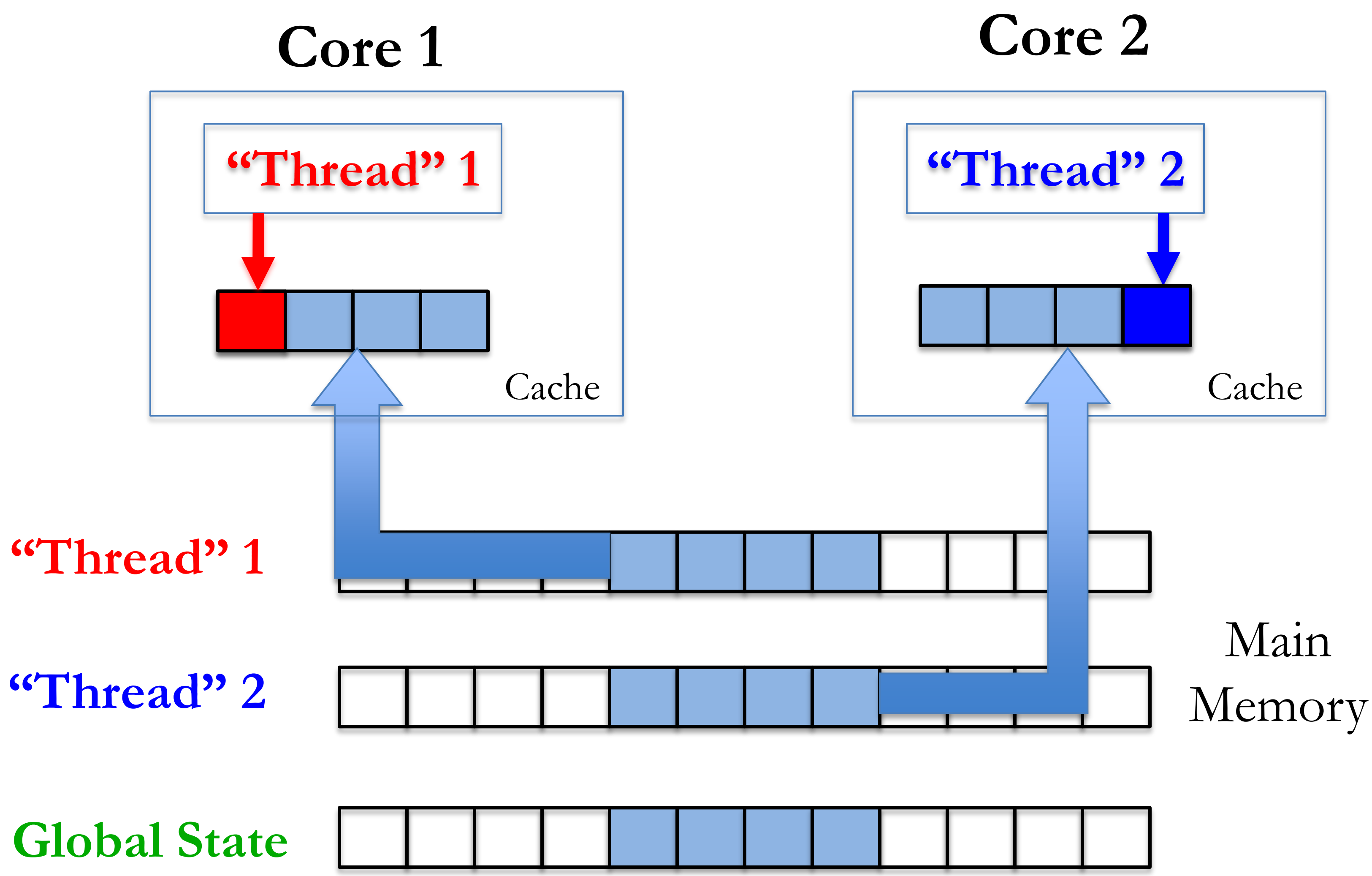
Sharing cache lines causes false sharing problems

Key Idea: Make Different Threads Access Different Cache Lines

Thread 1

Thread 2

Prevent False Sharing by Isolation



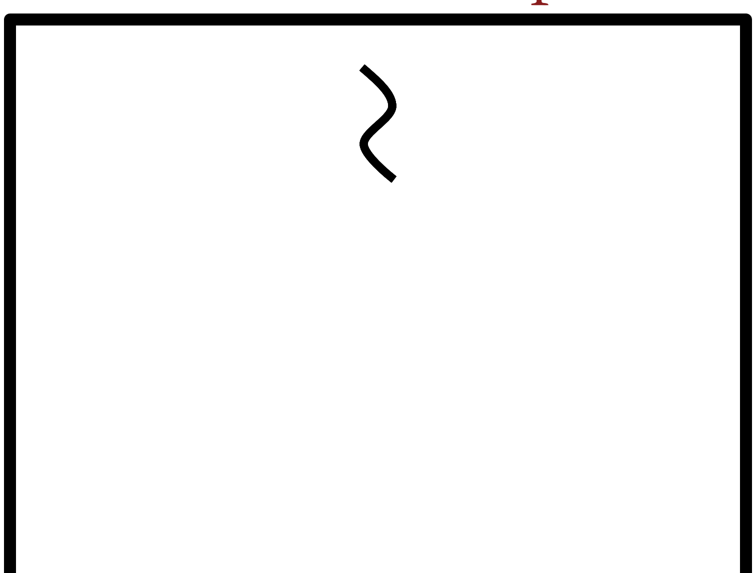
Processes-As-Threads

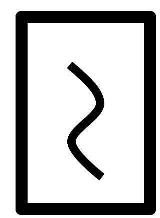
Threads

Processes

shared address space

disjoint address spaces





SHERIFF: Isolated Execution

Pthreads

1: Lock();

2: XX;

3: Unlock();

4: YY;

5: Lock();

SHERIFF

Lock_Process_Based();

Begin_isolated_execution
XX; *//isolated execution*
Commit_local_changes

Unlock_Process_Based();

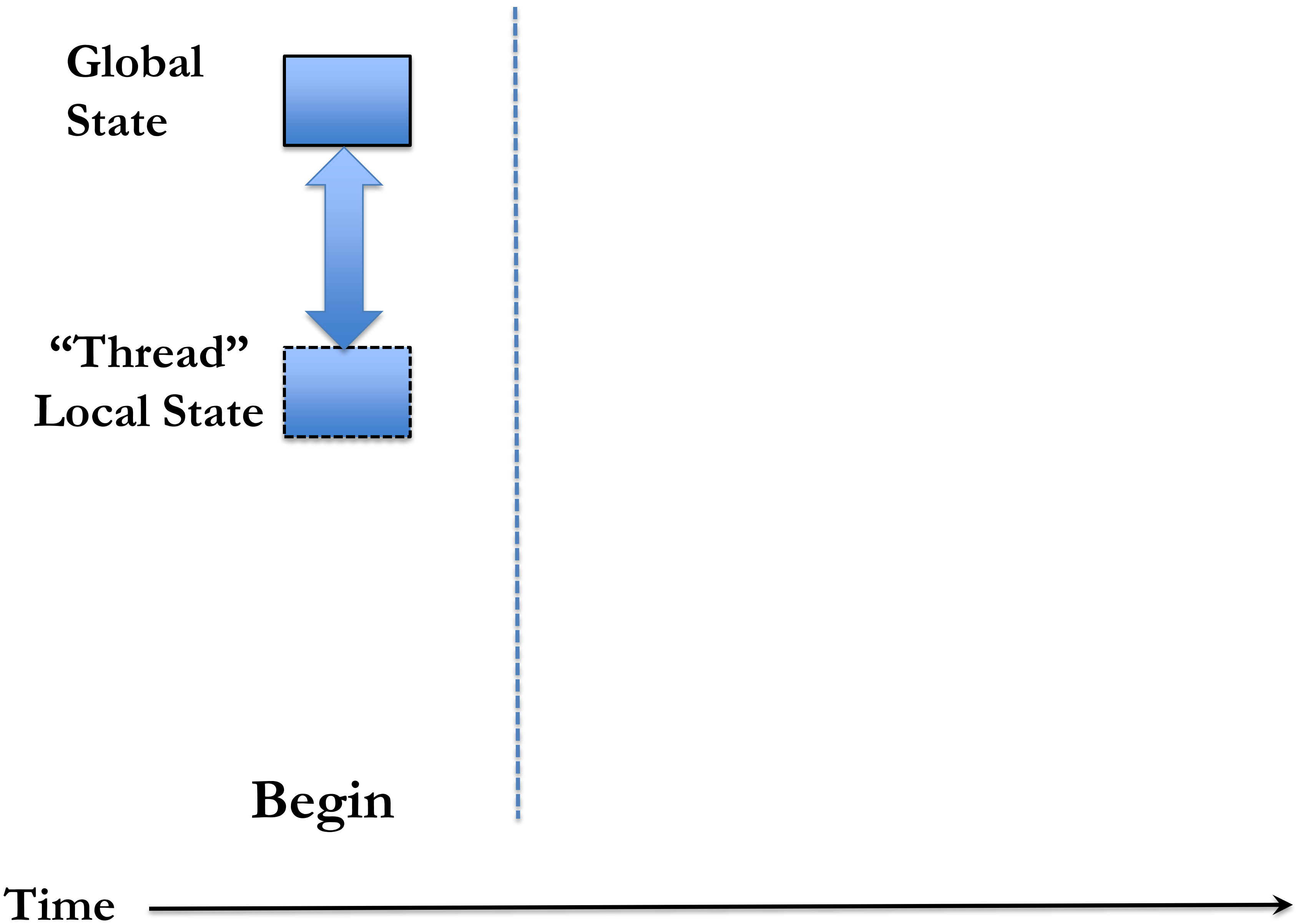
Begin_isolated_execution
YY; *//isolated execution*
Commit_local_changes

Lock_Process_Based();

Snapshot and Diffing: Find Local Changes

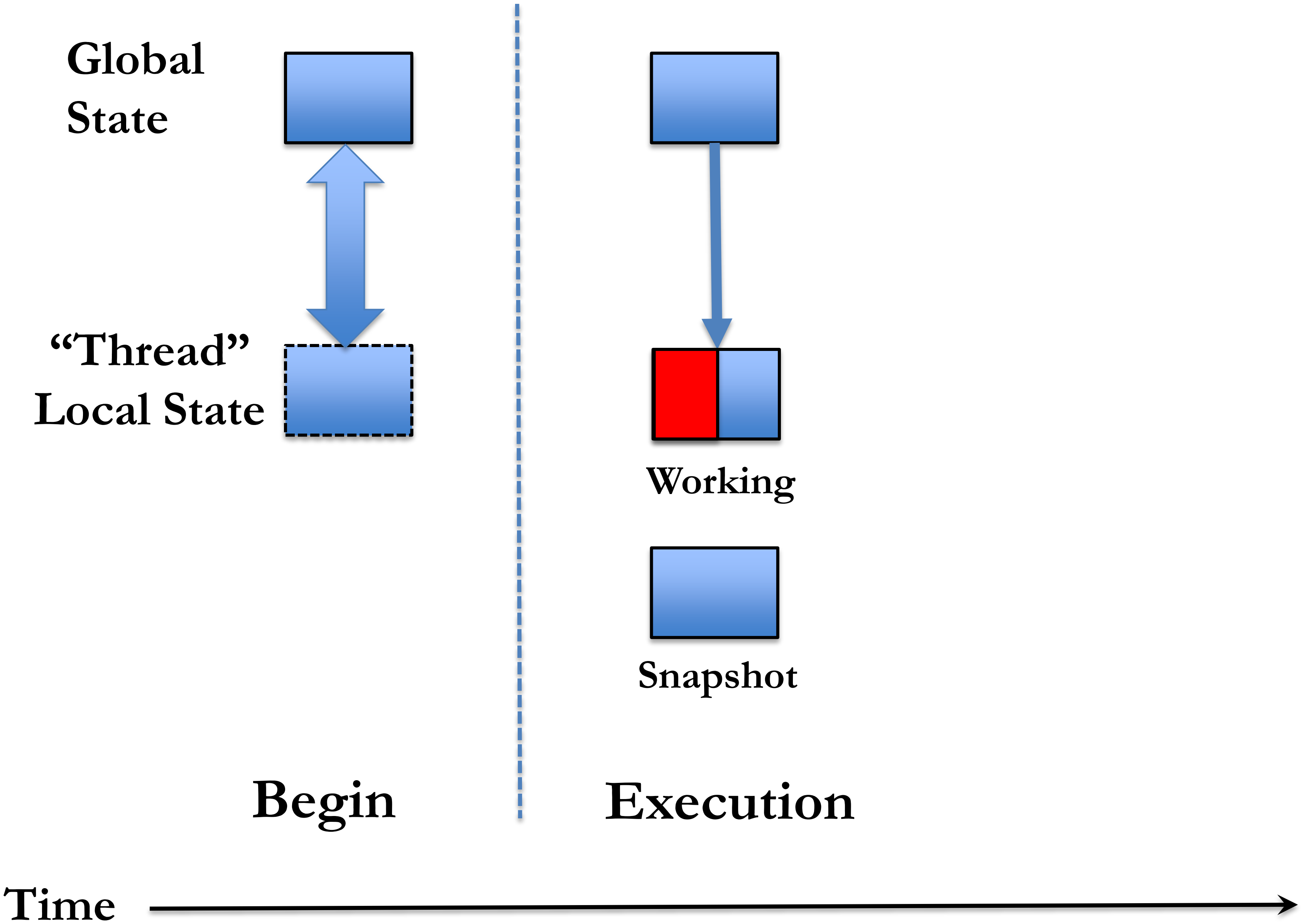
Snapshot

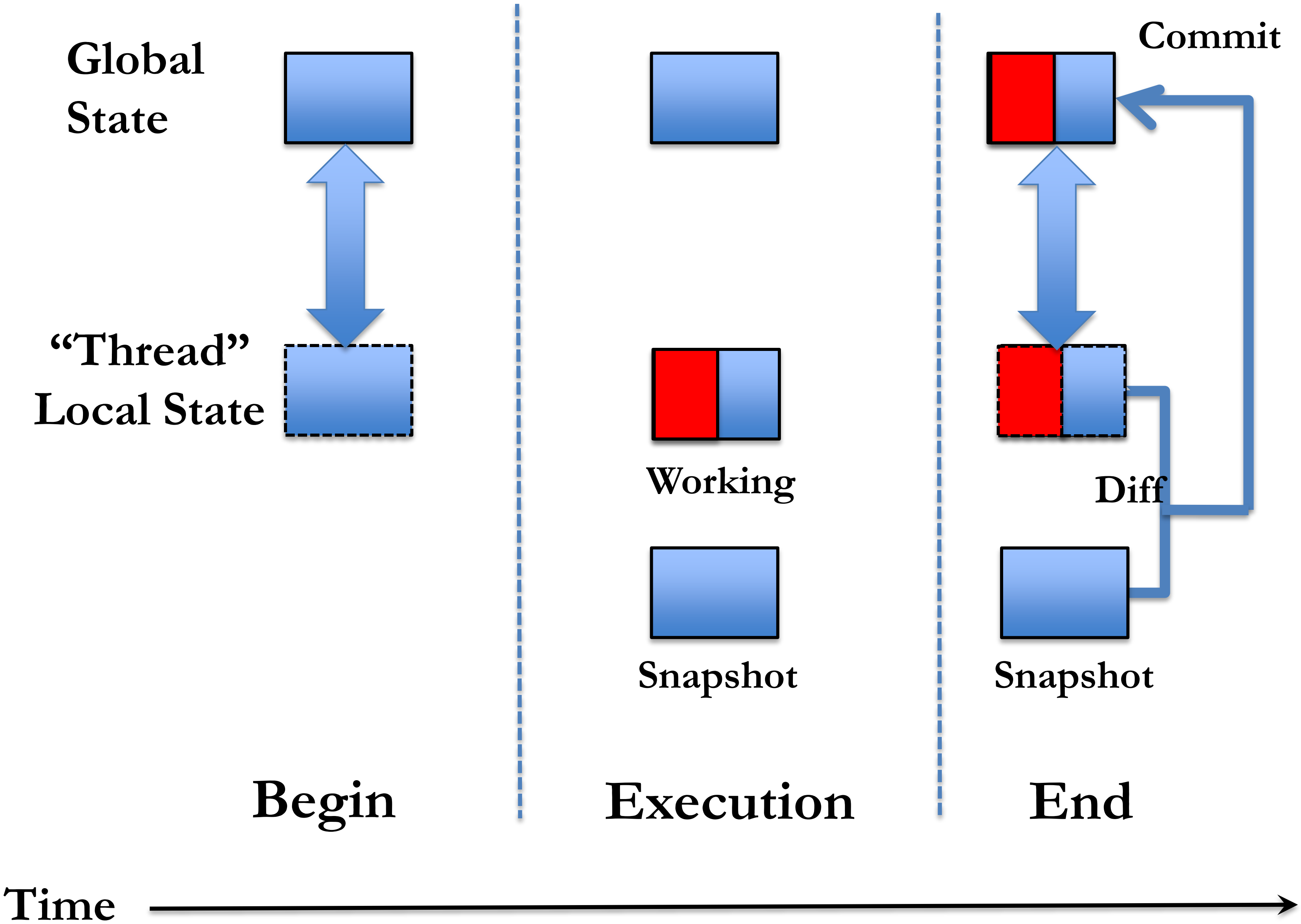
Working

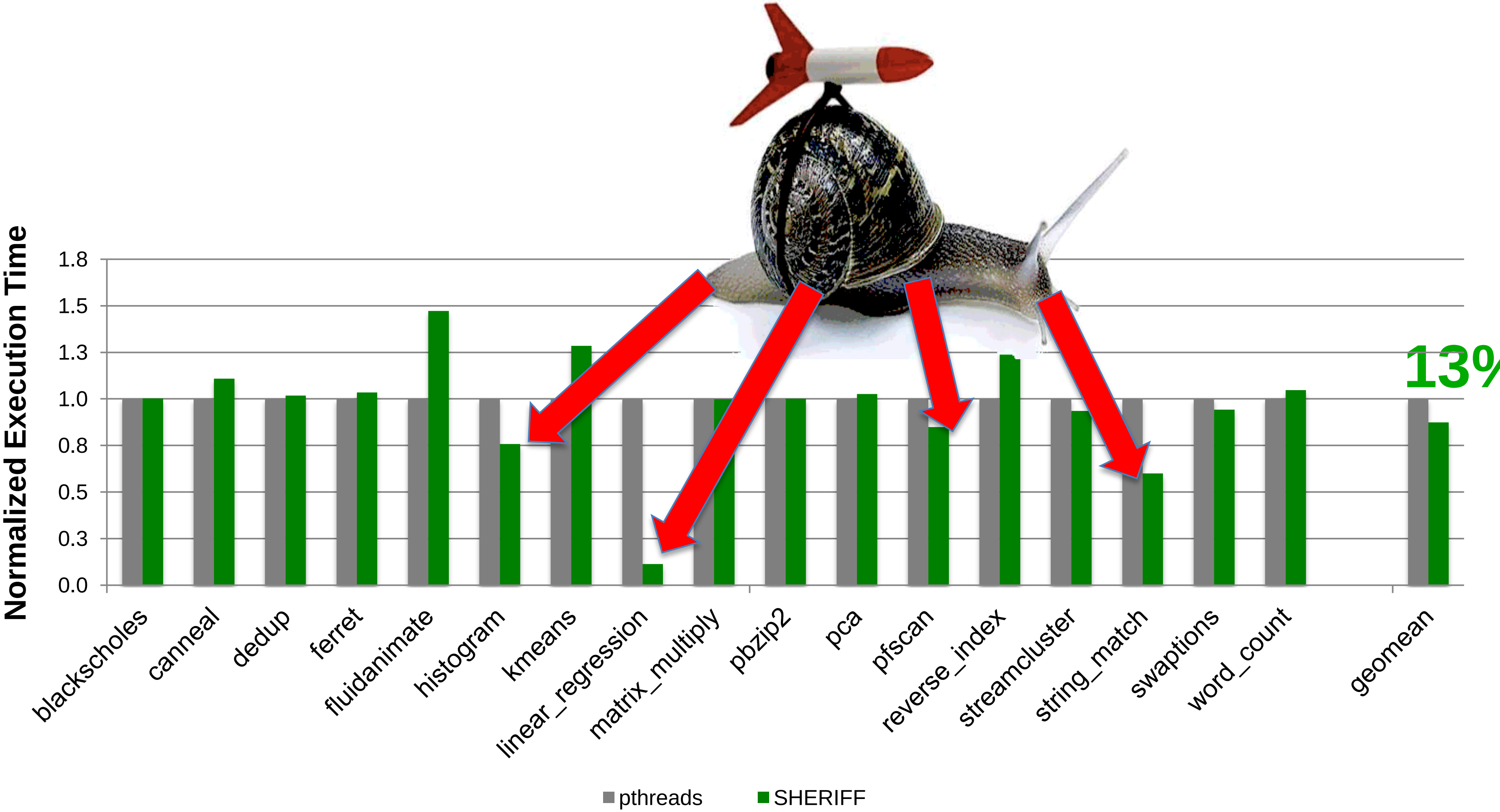

Global
State
"Thread"
Local State
Begin
Time

Detailed Memory Layout

- Local (private) ←→ global (shared): connect via file
- Applications only access thread local state (read-only initially, writable, read-only)


Global
State
“Thread”
Local State
Working
Snapshot
Begin
Execution
Time


Global
State
"Thread"
Local State
Working
Snapshot
Commit
Diff
Snapshot
Begin
Execution
End
Time



SHERIFF automatically boosts the performance of applications with false sharing

A Complete Solution for Parallel Applications with False Sharing

- First tool to pinpoint false sharing correctly and precisely
  - User can fix problems using padding or thread-local variables
- First generalized system to eliminate false sharing
  - Automatically boost performance without programmer intervention

Research Focus: Parallel Computing

Performance

SHERIFF: [Liu, OOPSLA'11]

Detecting and Tolerating False Sharing

PREDATOR: [Liu, PPOPP'14]

Predictive False Sharing Detection

Reliability

DTHREADS: [Liu, SOSP'11]

Efficient Deterministic Multithreading

DOUBLETAKE: [Liu, Submission]

Evidence-Triggered Dynamic Analysis

SOSP

23rd ACM Symposium on Operating Systems Principles
October 2011, Cascais, Portugal

DTHREADS:
Efficient Deterministic Multithreading

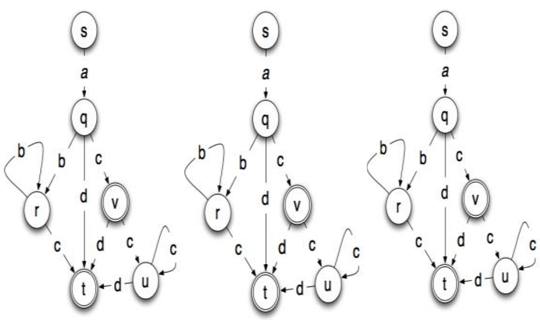
Tongping Liu, Charlie Curtsinger, Emery Berger

Citation: 101, 4th of 28 papers in SOSP 2011

DTHREADS Enables…

Deterministic executions

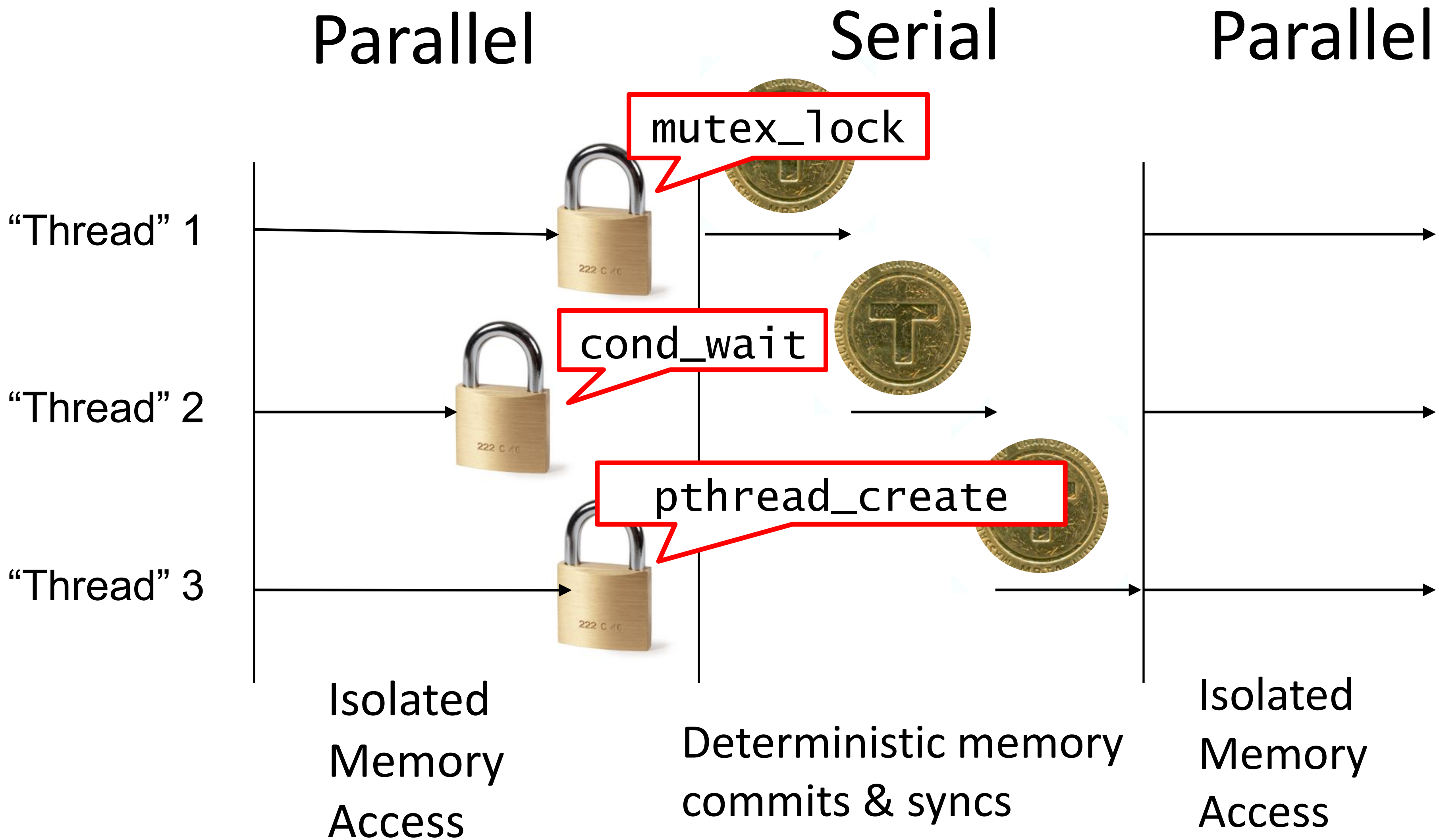
Replay w/o logging



Replicate applications on different machines

DTHREADS is the new basis of Deterministic Multithreading

Dthreads Overview



DoubleTake: Evidence-Triggered Dynamic Analysis

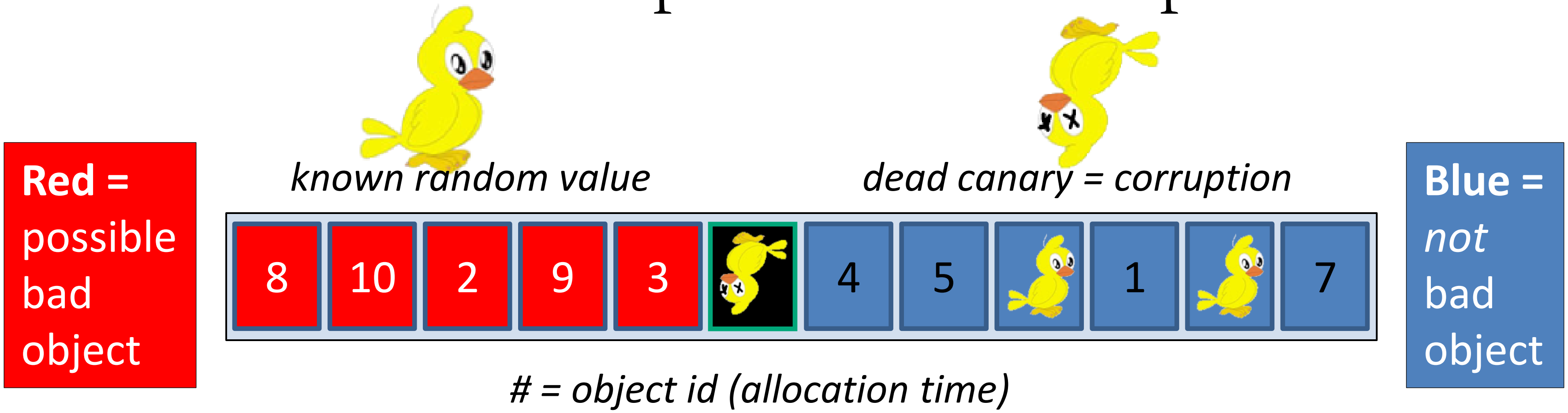
Tongping Liu, Charlie Curtsinger, Emery Berger

Heartbleed OpenSSL Problem:

"This vulnerability is due to a missing bounds check in the handling of the Transport Layer Security (TLS) heartbeat extension"

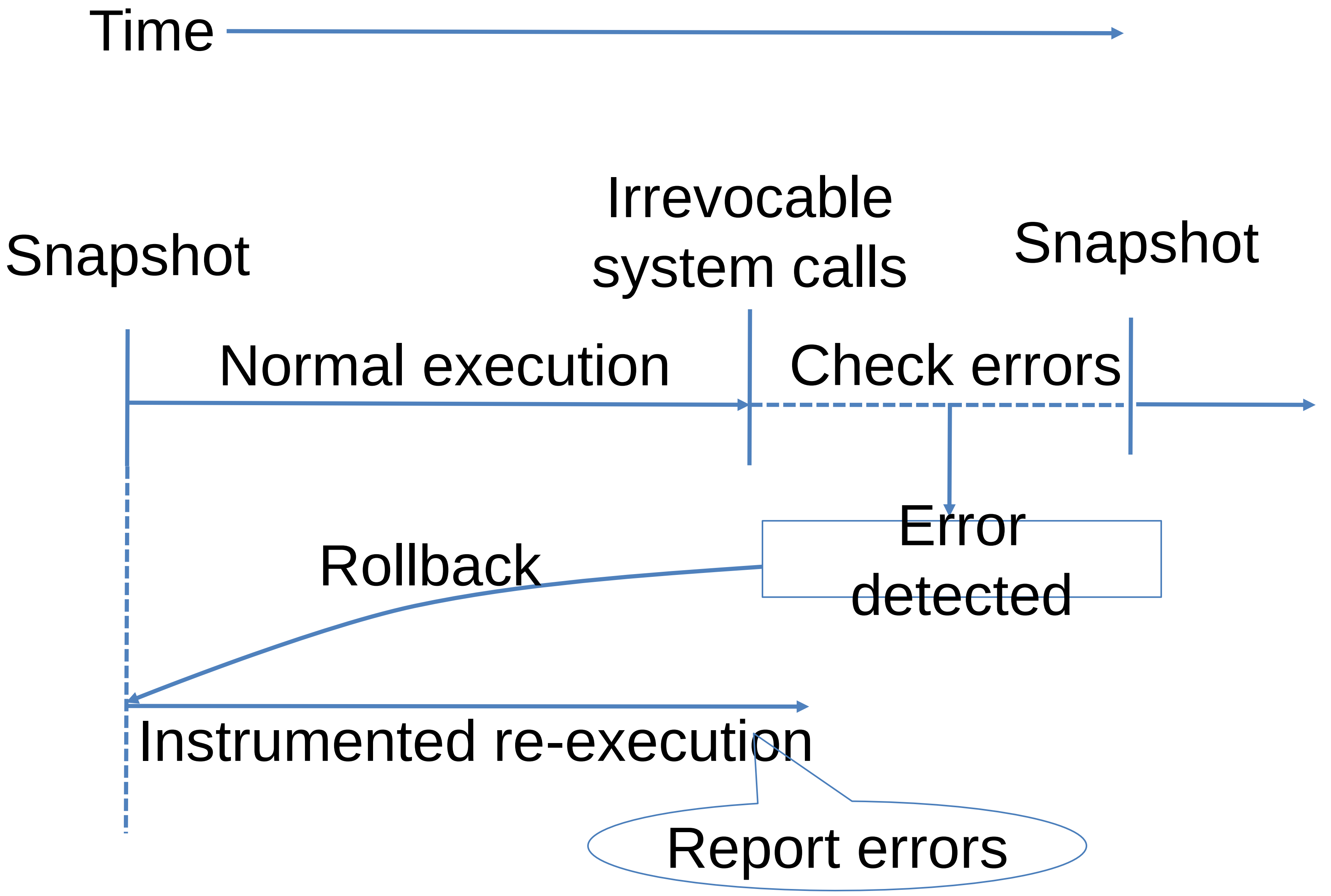
Detecting Buffer Overflows

- **Canaries** in freed space detect corruption

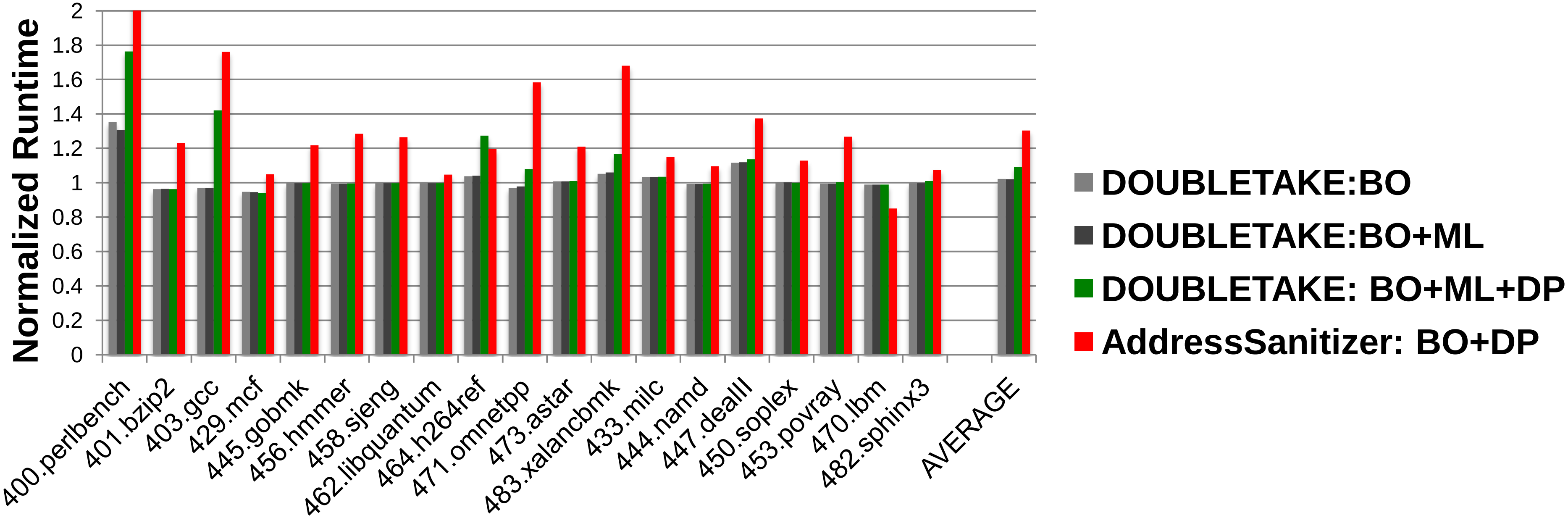


Precise detection:
instrument every memory read/write access

Imposing 33% overhead for common path!


Time
Snapshot
Irrevocable
system calls
Snapshot
Normal execution
Check errors
Error
detected
Rollback
Instrumented re-execution
Report errors

DOUBLETAKE: Efficient Memory Error Detection



BO: Buffer Overflow

BO+ML: only introduces 3% overhead
It is ready for the real deployment!

free)

Future Work (short-term)

- Complete false sharing solutions
  - Other languages: Java
  - Other software stacks: kernel, hypervisor
  - Improve performance using hardware-based approaches
  - Automatically fixes

- Other performance issues
  - lock granularity, thread model, scheduling

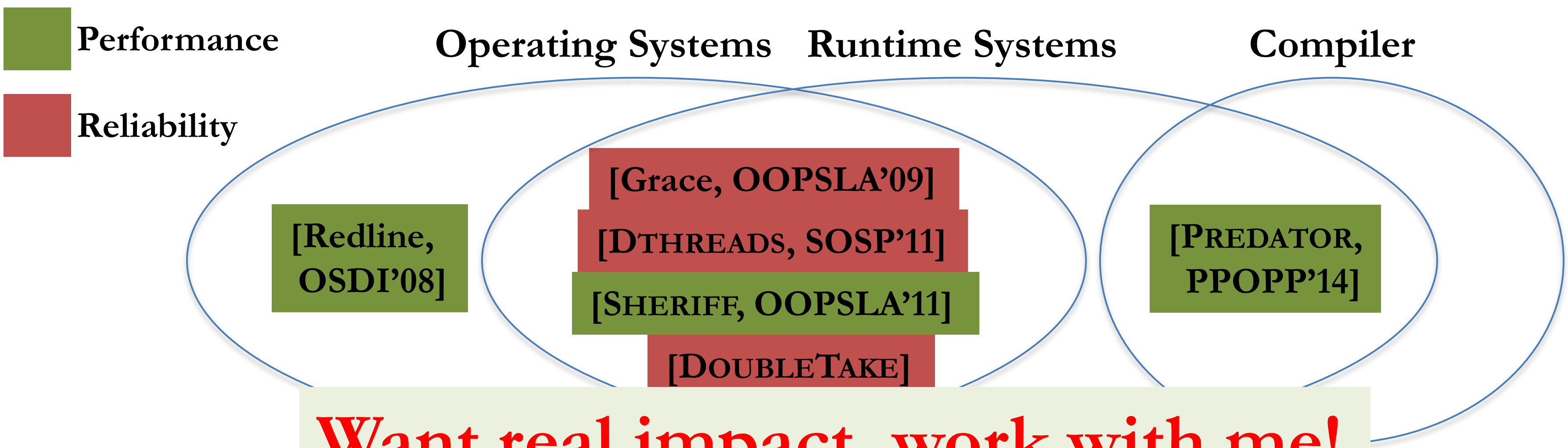
- Detect and prevent concurrency errors
  - Deadlocks, races, etc.

Future Work (long-term)

- Emerging hardware
  - NUMA: unpredictable performance, data sharing, efficient memory allocator
  - Heterogeneous systems (Start at NEC intern)
  - **Non-volatile memory**

- Cloud computing and BIG DATA systems
  - Quality of service (Related to Redline)
  - Performance of BIG DATA systems (Started at IBM intern)
  - Improve reliability (In study)
  - Energy efficiency (In study)

Conclusion and Future Work



Want real impact, work with me!

Main Contributions:

- ✧ Detecting and tolerating false sharing
- ✧ Efficiently detecting memory errors ✓ Done!
- ✧ New basis for deterministic multithreading

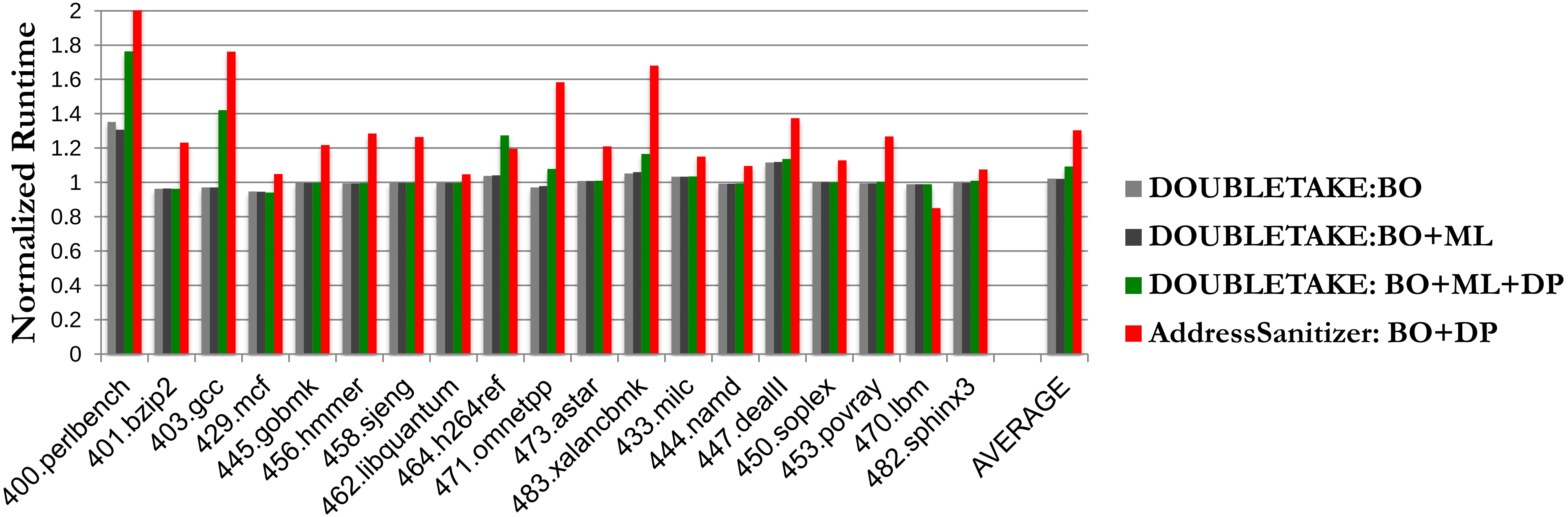
Future Work

1. False sharing
2. Concurrency errors
3. Other performance problems

1. Emerging hardware
2. Cloud computing
3. BIG DATA

Time

Performance Overhead of DOUBLETAKE



BO+ML: only introduces 3% overhead

It is ready for the real deployment.

Detailed Prediction Algorithm

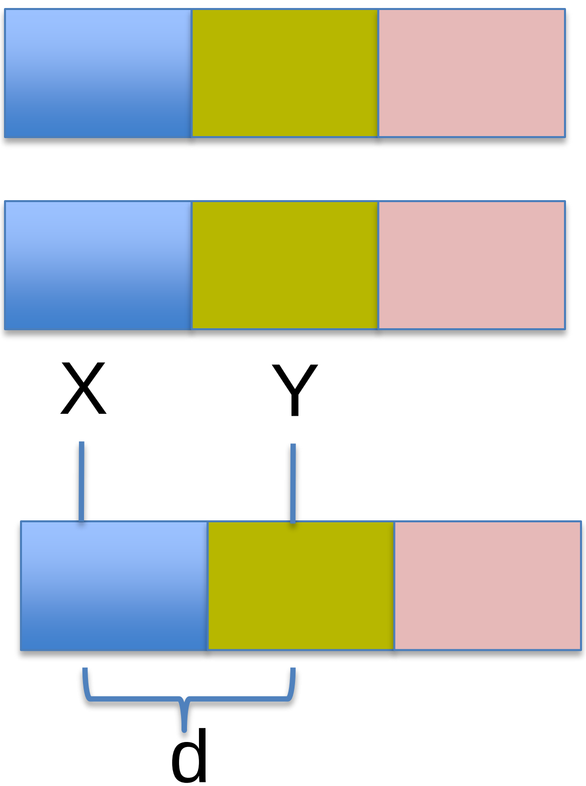
1. Find suspected cache lines

Detailed Prediction Algorithm

1. Find suspected cache lines

2. Track detailed memory accesses

Detailed Prediction Algorithm



1. Find suspected cache lines
2. Track detailed memory accesses
3. Predict based on hot accesses

d < sz && (X, Y) from different threads, potential false sharing

4: Tracking Cache Invalidations on the Virtual Line