CS636:Performance Analysis of Concurrent Programs

Swarnendu Biswas

Semester 2018-2019-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Evaluating an Application

Functional correctness

- Does the application compute/produce what it is supposed to do?

Performance correctness

- Does the application meet the performance requirements?

Testing for Performance

- No one wants slow and inefficient software
  - Leads to reduced throughput, increased latency, and wasted resources
  - Leads to poor UX

- Software efficiency is increasingly important
  - Hardware is not getting faster (per-core)
  - Software is getting more complex
  - Saving energy is now a primary concern

What is a Performance Bug?

Relatively simple modifications to the source code results in significant performance improvement, while preserving functionality

Functional and Performance Bugs

Functional Bugs	Performance Bugs
• Well-defined notion of success and failure	• Difficult to detect because of no failure symptoms
• Correctness requirements usually do not change over time • Other than significant changes in the specification	• Performance requirements may evolve over time
• More focus on researched testing methodologies	• Relative lack of formalized testing methodologies
• Rate of bugs generally flatten out with maturity	• Rate of bugs reported have no trends

Real Time Requirements

Hard

- Must meet the prescribed deadline, complete failure otherwise

Firm

- Can infrequently miss deadlines, the resultant computation might be useless, degrades QoS

Soft

- Might miss deadlines as long as the computation continues to have some value, degrades QoS

Characteristics of Performance Bugs

- Performance bugs can be **difficult** to fix
  - Contradictory requirements – a thread-safe class needs synchronization for correctness and needs to scale at the same time
  - Diminishing returns in fixing performance bugs

Thread Safe (?) Class from Groovy

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.initialized = true;
  }
  boolean isInitialized() {
    return this.initialized;
  }
}
```

M. Pradel et al. Performance Regression Testing of Concurrent Classes. ISSTA 2014.

Fixing a Thread Safe Class from Groovy

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.initialized = true;
  }
  boolean isInitialized() {
    return this.initialized;
  }
}
```

Before October 2007: Class is not thread-safe because reads and writes of `initialized` are not synchronized

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.setInitialized(true);
  }
  synchronized void setInitialized(boolean b) {
    this.initialized = b;
  }
  synchronized boolean isInitialized() {
    return this.initialized;
  }
}
```

October 2007: Fixed thread safety problem by making methods synchronized. Led to performance regression reported in May 2009.

Fixing a Thread Safe Class from Groovy

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.initialized = true;
  }
  boolean isInitialized() {
    return this.initialized;
  }
}
```

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.setInitialized(true);
  }
  synchronized void setInitialized(boolean b) {
    this.initialized = b;
  }
  synchronized boolean isInitialized() {
    return this.initialized;
  }
}
```

```
class ExpandoMetaClass {
  private volatile boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.setInitialized(true);
  }
  void setInitialized(boolean b) { this.initialized = b; }
  boolean isInitialized() { return this.initialized; }
}
```

September 2009: Fixed performance regression by replacing synchronized methods with volatile variables

Fixing a Thread Safe Class from Groovy

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.initialized = true;
  }
  boolean isInitialized() {
    return this.initialized;
  }
}
```

```
class ExpandoMetaClass {
  private boolean initialized;
  synchronized void initialize() {
    if (!this.initialized)
      this.setInitialized(true);
  }
  synchronized void setInitialized(boolean b) {
    this.initialized = b;
```

Why have the `synchronized` keyword then? Wouldn't `volatile` suffice?

```
class ExpandoMeta
  private volatil

  synchronized voi
    if (!this.initialized)
      this.setInitialized(true);
  }
  void setInitialized(boolean b) { this.initialized = b; }
  boolean isInitialized() { return this.initialized; }
}
```

September 2009: Fixed performance regression by replacing synchronized methods with volatile variables

Example 1

Apache HTTPD developers forgot to change a parameter of API `apr_stat` after an API upgrade. This mistake caused more than ten times slowdown in Apache servers.

Patch for Apache Bug 45464

modules/dav/fs/repos.c

```
status = apr_stat ( fscontext->info,
-                   APR_DEFAULT);
+                    APR_TYPE);
```

What is this bug

An Apache-API upgrade causes **apr_stat** to retrieve more information from the file system and take longer time.

Now, APR_TYPE retrieves exactly what developers originally needed through APR_DEFAULT.

Impact: causes httpd server 10+ times slower in file listing

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Example 2

Mozilla developers implemented a procedure `nsImage::Draw` for figure scaling, compositing, and rendering, which is a waste of time for transparent figures. This problem did not catch developers' attention until two years later when 1 pixel by 1 pixel transparent GIFs became general purpose spacers widely used by Web developers to work around certain idiosyncrasies in HTML 4. The patch of this bug skips `nsImage::Draw` when the function input is a transparent figure.

Mozilla Bug 66461 & Patch

```
nsImage::Draw(...) {
    ...
+ if(mIsTransparent) return;
    ...
  //render the input image
}
```

nsImageGTK.cpp

What is this bug

When the input is a transparent image, all the computation in *Draw* is useless.

Mozilla developers did not expect that transparent images are commonly used by web developers to help layout.

The patch conditionally skips *Draw*.

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Example 3

Users reported that Firefox cost 10 times more CPU than Safari on some popular Web pages, such as gmail.com. Lengthy profiling and code investigation revealed that Firefox conducted an expensive garbage collection process GC at the end of every XMLHttpRequest, which is too frequent. A developer then recalled that GC was added there five years ago when XHRs were infrequent and each XHR replaced substantial portions of the DOM in JavaScript. However, things have changed in modern Web pages. As a primary feature enabling web 2.0, XHRs are much more common than five years ago. This bug is fixed by removing the call to GC.

Mozilla Bug 515287 & Patch

```
XMLHttpRequest::OnStop(){
  //at the end of each XHR
  ...
-- mScriptContext->GC();
}
```

nsXMLHttpRequest.cpp

What is this bug

This was not a bug until Web 2.0, where doing garbage collection (*GC*) after every XMLHttpRequest (XHR) is too frequent.

It causes Firefox to consume **10X more CPU** at idle GMail pages than Safari.

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Example 4

Users reported that Firefox hung when they clicked 'bookmark all (tabs)' with 20 open tabs. Investigation revealed that Firefox used N database transactions to bookmark N tabs, which is very time consuming comparing with batching all bookmark tasks into a single transaction. Discussion among developers revealed that the database service library of Firefox did not provide interface for aggregating tasks into one transaction, because there was almost no batchable database task in Firefox a few years back. The addition of batchable functionalities such as 'bookmark all (tabs)' exposed this inefficiency problem. After replacing N invocations of `doTransact` with a single `doAggregateTransact`, the hang disappears. During patch review, developers found two more places with similar problems and fixed them by `doAggregateTransact`.

Mozilla Bug 490742 & Patch

```
for (i = 0; i < tabs.length; i++) {
    ...
-     tabs[i].doTransact();
}
+ doAggregateTransact(tabs);
```

nsPlacesTransactionsService.js

What is this bug

doTransact saves one tab into 'bookmark' SQLite Database.

Firefox hangs @ 'bookmark all (tabs)'.

The patch adds a new API to aggregate DB transactions.

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Example 5

MySQL synchronization-library developers implemented a `fastmutex_lock` for fast locking. Unfortunately, users' unit test showed that `fastmutex_lock` could be 40 times slower than normal locks. It turns out that library function `random()` actually contains a lock. This lock serializes every threads that invoke `random()`. Developers fixed this bug by replacing `random()` with a non-synchronized random number generator.

MySQL Bug 38941 & Patch

```
int fastmutex_lock (fmutex_t *mp){
  ...
-  maxdelay += (double) random();
+  maxdelay += (double) park_rng();
  ...
}
```

thr_mutex.c

What is this bug

random() is a serialized global-mutex-protected glibc function.

Using it inside `*fastmutex*` causes **40X slowdown** in users' experiments.

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Performance Bugs are Surprisingly Common!

	Type	Language	# Bugs
Apache	Command-line utility + Server + Library	C/Java	25
Google Chrome	GUI Application	C/C++	10
GCC	Compiler	C/C++	10
Mozilla	GUI Application	C++/JS	36
MySQL	Server Software	C/C++/C#	28

G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.

Reasons for Performance Bugs

Inefficient function call combinations

Redundant work

Resource contention (e.g., suboptimal synchronization, false sharing)

- Many synchronization fixes are just because of performance reasons

Cross core/node communication

Miscellaneous

- Poor data structure choices
- Design/algorithm issues

How Performance Bugs are Introduced?

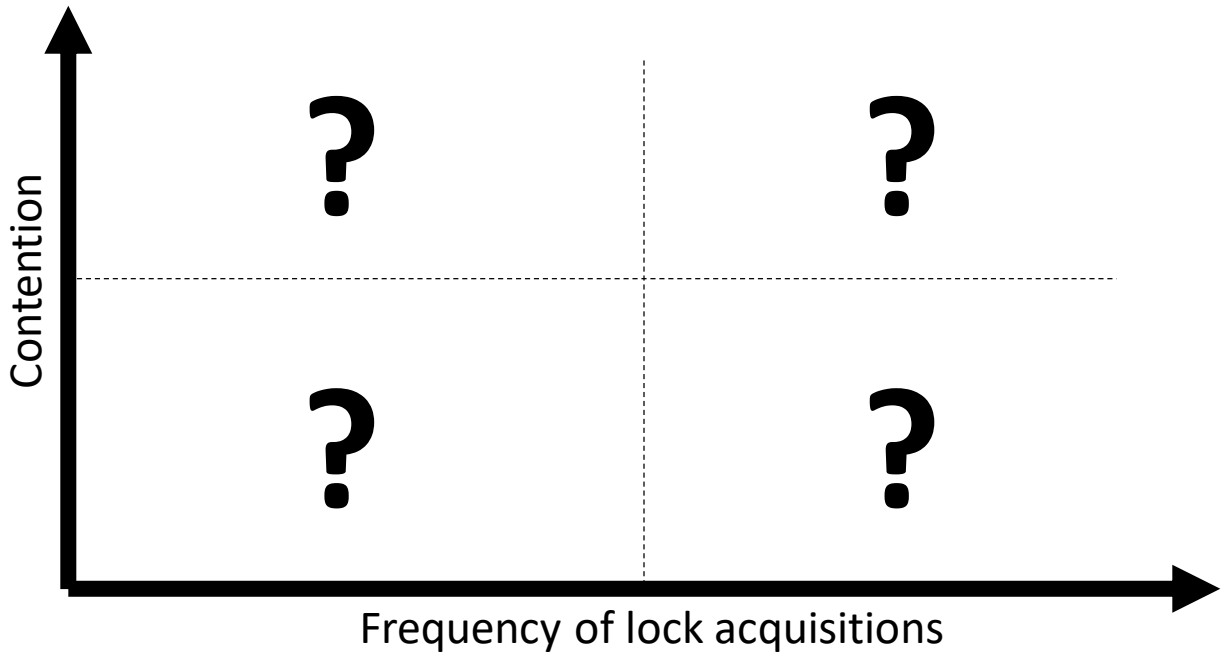
- Mismatch in workload characterization
- Wrong API interpretation
- Miscellaneous
  - Wrong functional implementation – leads to redundant work
- Changes in performance requirements

Dealing with Performance Bugs

- Compilers and hardware optimizations may not always fix performance problems
- Automation support is limited and is still being explored
  - Design better performance tests
  - Use annotation systems
  - …

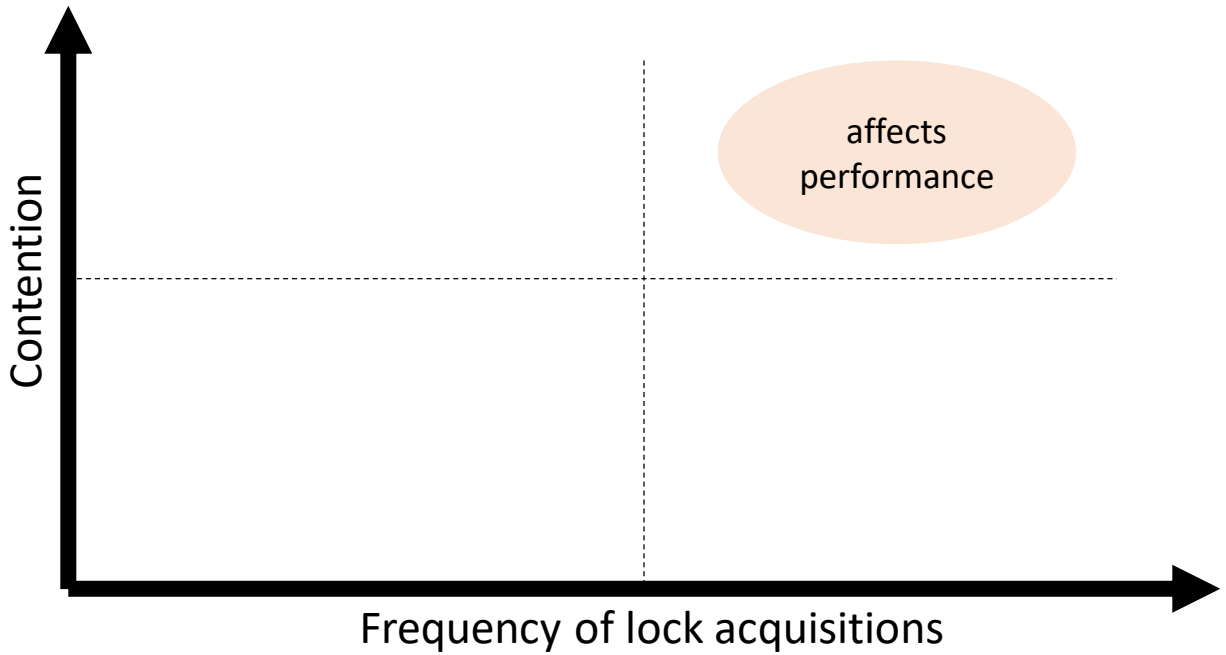
Tracking Synchronization Bottlenecks

Synchronization-Related Factors That Affect Performance



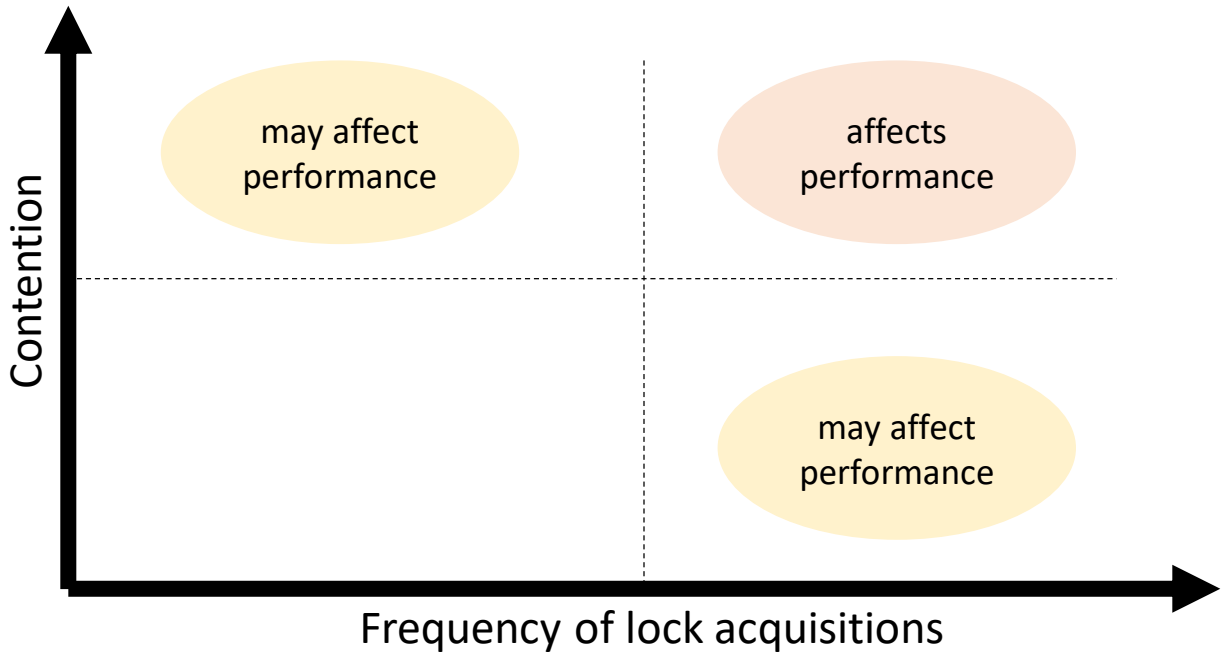
M. Alam et al. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. EuroSys 2017.

Synchronization-Related Factors That Affect Performance



M. Alam et al. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. EuroSys 2017.

Synchronization-Related Factors That Affect Performance



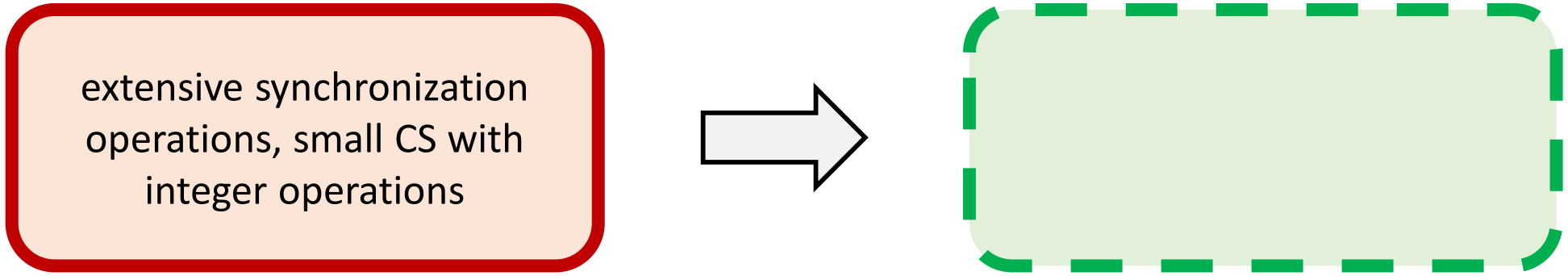
M. Alam et al. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. EuroSys 2017.

Spectrum of Synchronization Operations

Type of Synchronization	Ideal Use Case
atomic instructions	simple integer operations (RMW, exchange)
spin locks	small critical sections with low contention
read/write locks	critical sections with many readers
try locks	alternate control flow
mutex locks	larger critical sections and may involve waiting

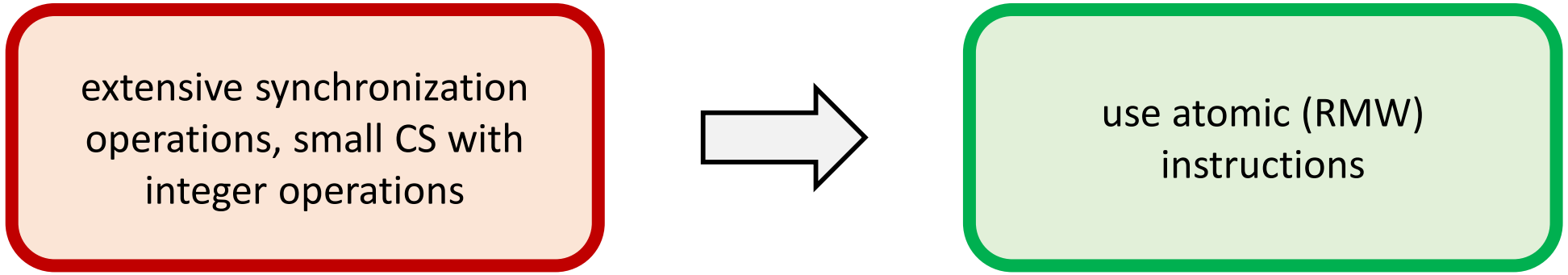
Reasons for Synchronization-Related Performance Loss

- Use of improper primitives
  - E.g., use of try locks in case of repeated failures, blocking synchronization with condition variables might be better



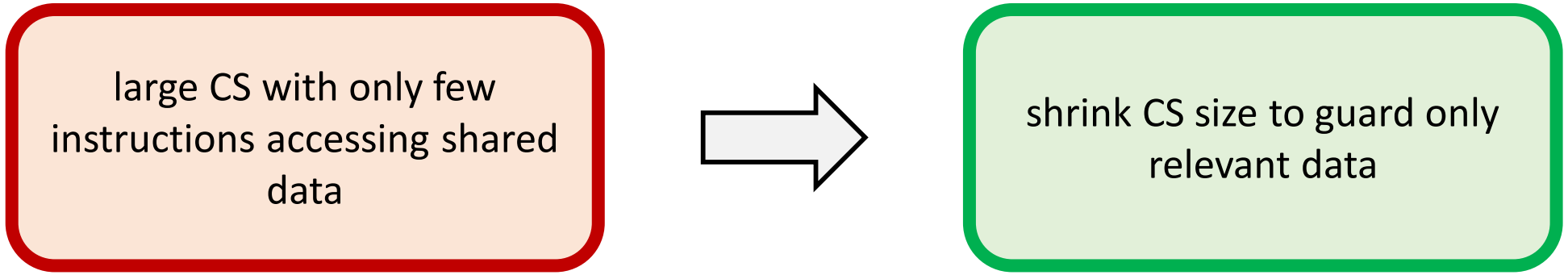
Reasons for Synchronization-Related Performance Loss

- Use of improper primitives
  - E.g., use of try locks in case of repeated failures, blocking synchronization with condition variables might be better



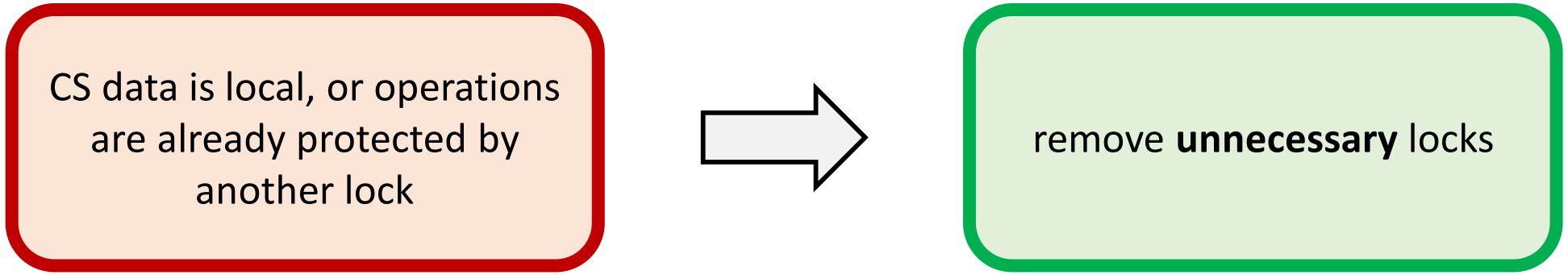
Reasons for Synchronization-Related Performance Loss

- Wrong granularity choice
  - E.g., look out for refining coarse locks into finer-grained locks



Reasons for Synchronization-Related Performance Loss

- Over synchronization
  - CS data is thread-local or read-only or may write to disjoint addresses
  - Operations are already protected by another lock



Reasons for Synchronization-Related Performance Loss

- Asymmetric contention
  - E.g., say a poor hash function fails to distribute items to different buckets and locks are taken per bucket
- Load imbalance
  - Waiting time for a group of threads is more than for other group(s) of threads

Automated Analyses for Detecting Synchronization-Related Performance Bugs

- Lock contention detectors
  - Thread Profiler, IBM Lock Analyzer, SyncProf, …
  - Measure thread idle time, thread synchronization time
- Study impact of critical sections on the critical paths of applications
  - Focus on locks that can impact performance
- Detect load imbalance
- General profiling tools

Speculative Lock Elision

Potential Parallelism Hurt by Synchronization

```
LOCK(locks->error_lock);
if (local_error > multi->err_multi)
  multi->err_multi = local_error;
UNLOCK(locks->error_lock);
```

R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. MICRO 2001.

Problems with Conservative Locking

- Many lock operations are not necessary
  - Updates in the critical sections occur infrequently during execution
  - Updates can occur to disjoint parts of the data structure

- Conventional speculative execution in OOO processors are not able to deal with these situations

Speculative Lock Elision (SLE)

Idea

- Speculatively assume lock is not necessary and execute critical section without acquiring the lock
- Check for conflicts within the critical section
- Roll back if assumption is incorrect

SLE can be provided with both software and hardware support

Challenges in Providing SLE

Either the entire critical section is committed or none of it

Challenges

- How to detect the lock operation that is to be elided?
- How to keep track of dependences and conflicts in the critical section?
- How to buffer speculative state?
- How to check if “atomicity” is violated?
  - That is, check for conflicts
- How to support commit and rollback?

Maintaining Atomicity

- If atomicity is maintained, all locks can be removed
- Conditions for atomicity
  - Data read is not modified by another thread until critical section is complete
  - Data written is not accessed by another thread until critical section is complete
- If we know the beginning and end of a critical section, we can monitor the memory addresses read or written to by the critical section and check for conflicts
  - For example, using the underlying coherence mechanism

Potential SLE Implementation in Hardware

- Checkpoint register state before entering SLE mode
- In SLE mode
  - Store: Buffer the update in the write buffer (invisible to other processors), request exclusive access
  - Store/Load: Set “access” bit for block in the cache
  - Trigger misspeculation on some coherence actions
    - If external invalidation to a block with “access” bit set
    - If exclusive access to request to a block with “access” bit set
  - If not enough buffering space, trigger misspeculation
- If end of critical section reached without misspeculation, commit all writes (needs to appear instantaneous)

When is SLE Advantageous?

There is little contention between the critical section operations from concurrent threads

Why?

Threads contend for the lock protecting the critical section

Why?

Expected Gains from SLE

Concurrent critical section execution

Reduced memory latencies to lock locations

- Lock memory locations can remain shared

Reduced memory traffic

- No transfer of coherence messages over the bus

Limitations

- Hardware implementation is constrained by the size of the cache and the write buffers

SLE vs TM

SLE	TM
• Track memory accesses in critical sections, detect conflicts, and perform rollbacks	• Track memory accesses in transactions, detect conflicts, and perform rollbacks
• “Best effort” – can fallback to acquire the lock and reexecute non-speculatively	• TM generally is always speculative
• Need to identify opportunities for lock elision	• Complete program execution can be transactional

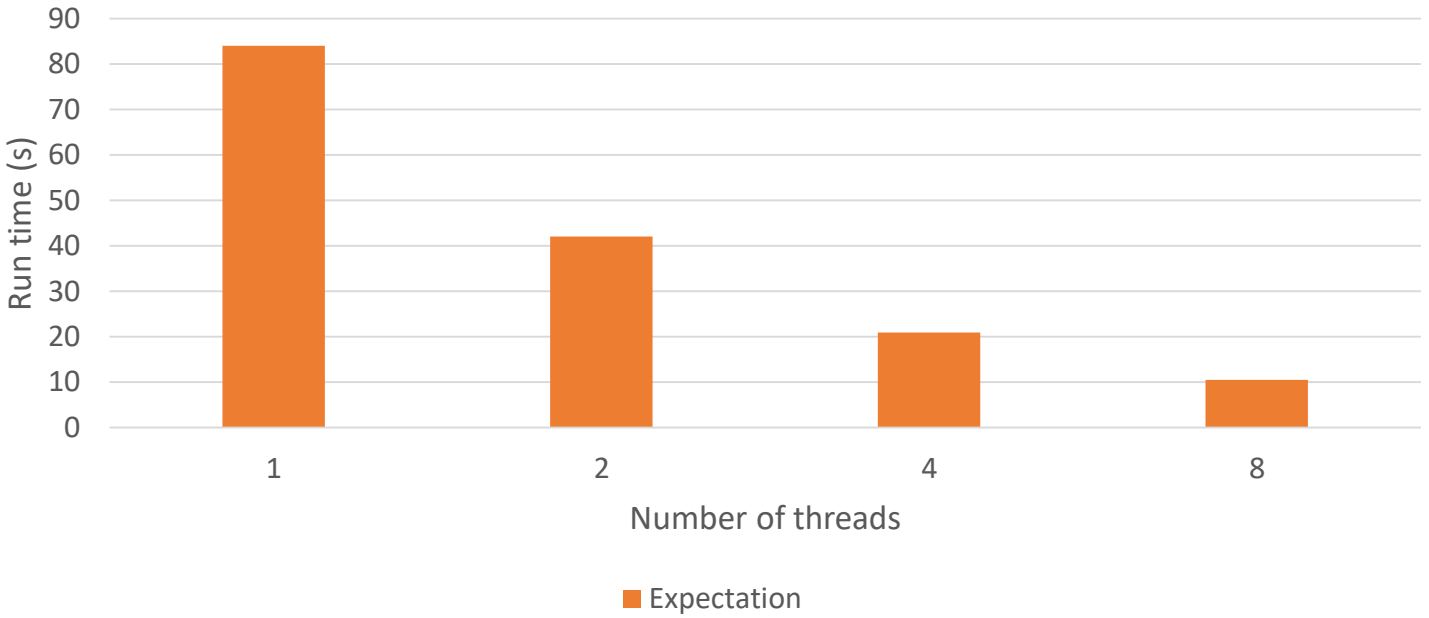
Dealing with False Sharing

Multicore Parallelism is Easy

```
int count[8]; // Global array

thread_func(int id) {
  for(i = 0; i < M; i++)
    count[id]++;
}
```

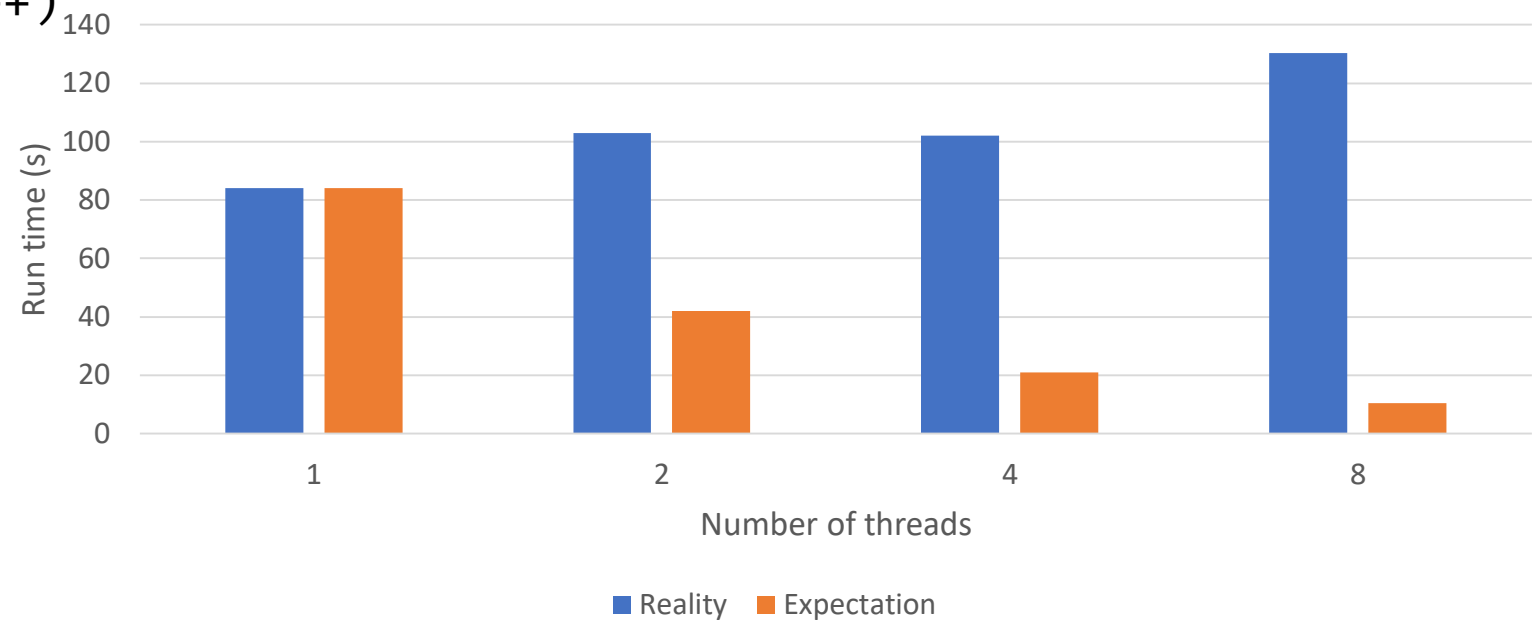
We are expecting strong scaling



Multicore Parallelism is Easy

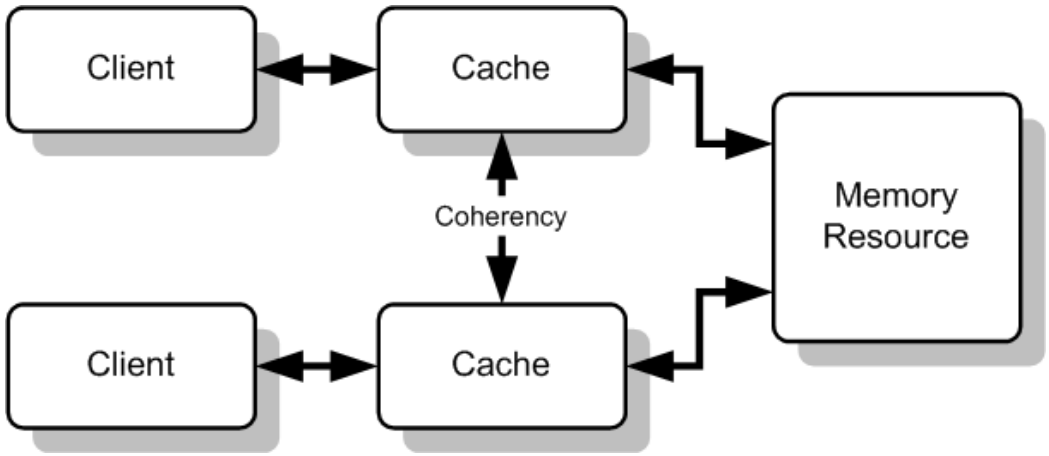
```
int count[8]; // Global array

thread_func(int id) {
  for(i = 0; i < M; i++)
    count[id]++;
}
```

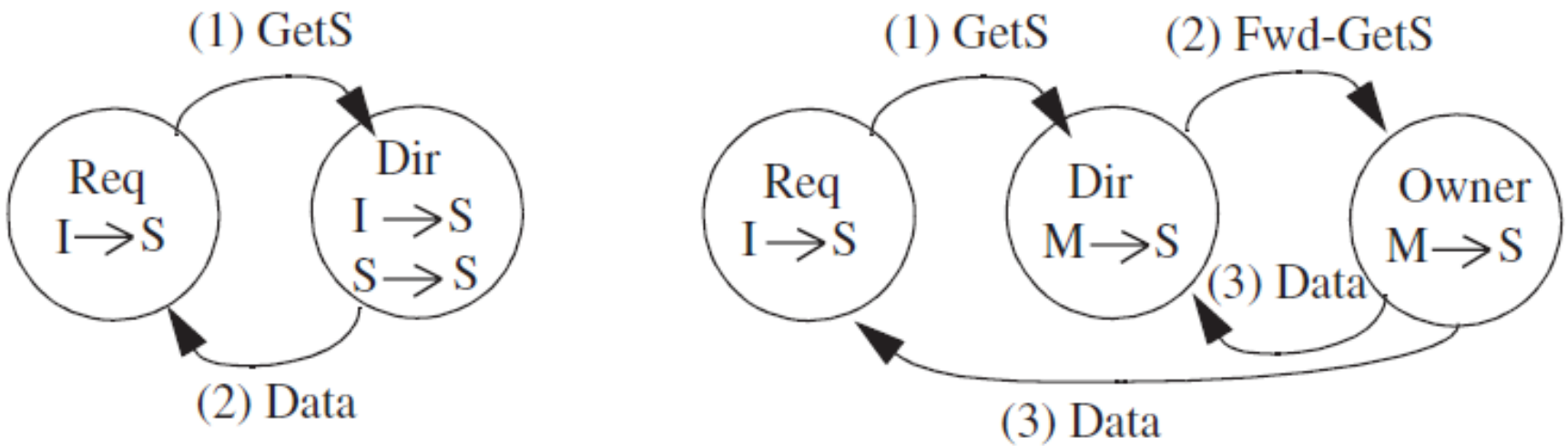


Cache Coherence

- Multicore processors implement a cache coherence protocol to keep private caches in sync
- Operates on whole cache lines (usually 64 bytes)
- Cache lines have three key states:
  - Read Shared (S), Write Exclusive (M), Invalid (I)



MSI Directory Protocol



Transitions from I to S.

Fig 8.3 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

MSI Directory Protocol

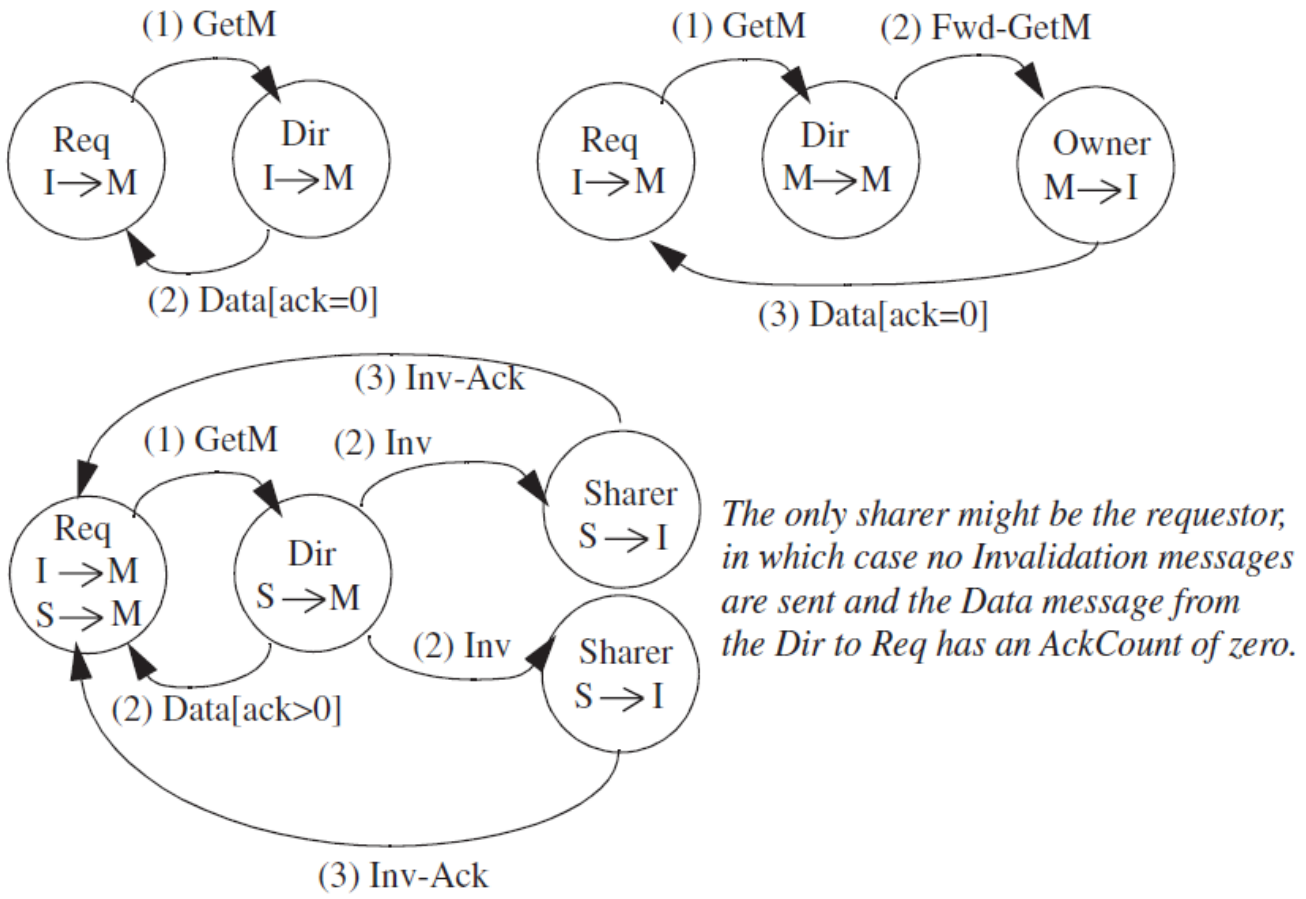


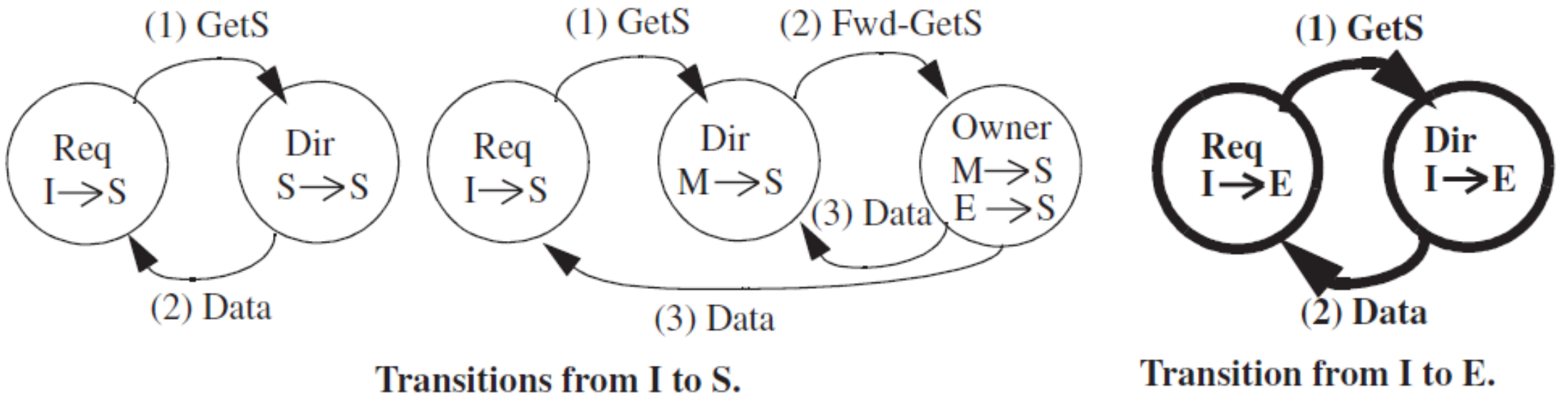
Fig 8.3 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

MSI Directory Protocol

Transition from M or S to I

Fig 8.3 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

MESI Directory Protocol

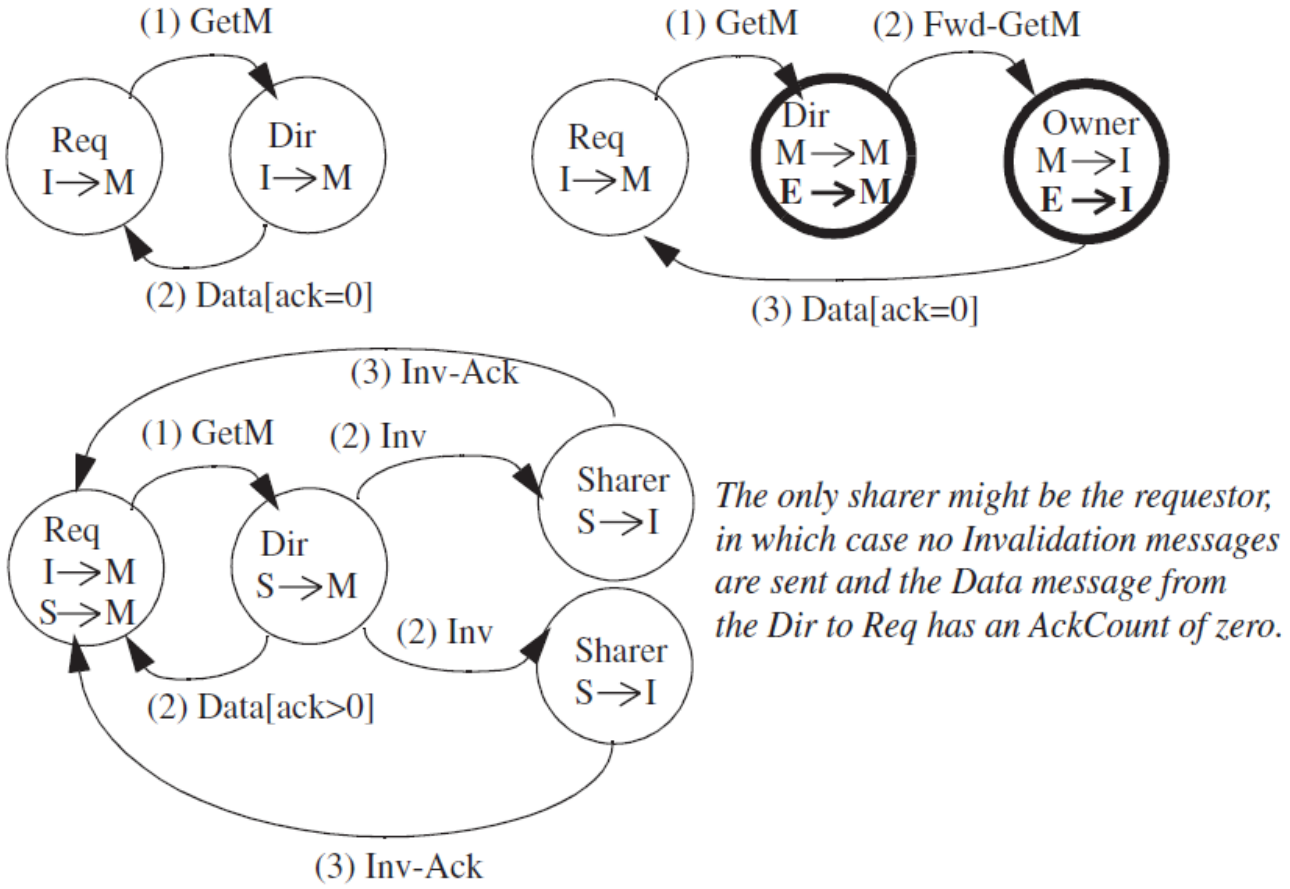


Transitions from I to S.

Transition from I to E.

Fig 8.6 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

MESI Directory Protocol



The only sharer might be the requestor, in which case no Invalidation messages are sent and the Data message from the Dir to Req has an AckCount of zero.

Transitions from I or S to M. Transition from E to M is silent.

Fig 8.6 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

MESI Directory Protocol

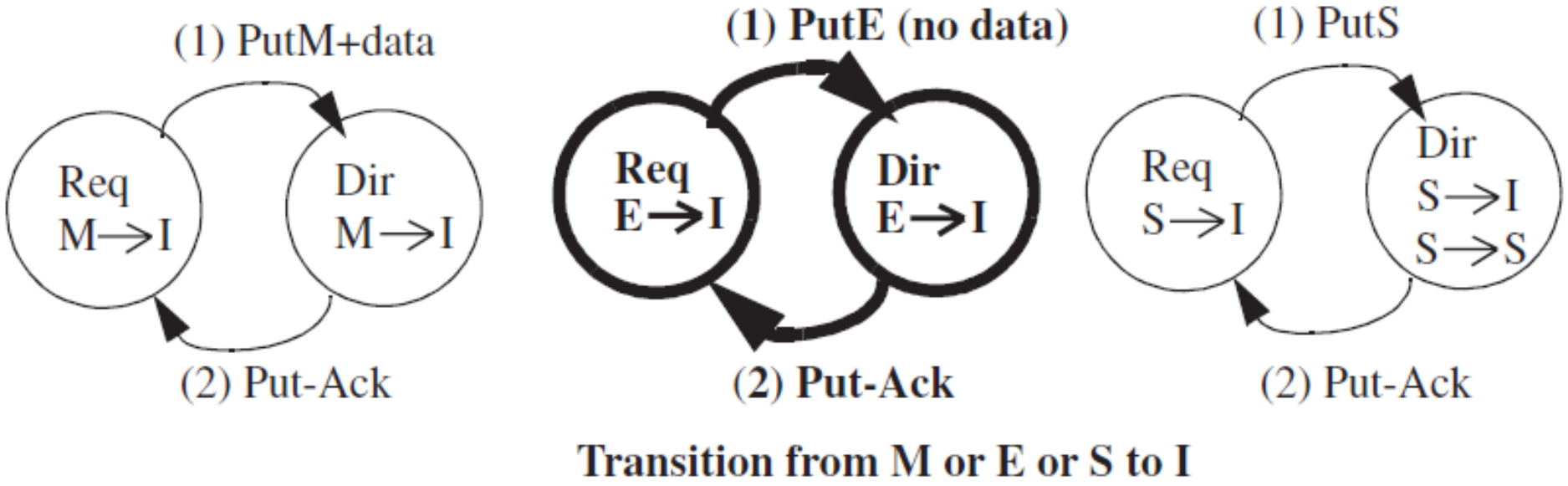


Fig 8.6 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

What is False Sharing?

- Performance problem in systems with coherence caches
  - Cores share cache blocks instead of actual data
  - Contention on cache lines
- Can arise when threads access global or heap memory
  - Thread-local storage and local variables can be ignored
- False sharing is aggravated by the size of cache block

Cache Contention

True Sharing

- Fixed by means of application restructuring

False Sharing

- Fixed by code changes or by automatic repair

Impact of False Sharing

```
int array[100];

void *func(void *param) {
  int   index = *((int*)param);
  int   i;
  for (i = 0; i < 100000000; i++)
    array[index]+=1;
}

int main(int argc, char *argv[]) {
  int       first_elem  = 0;
  int       bad_elem    = 1;
  int       good_elem   = 99;

  pthread_t thread_1;
  pthread_t thread_2;
```

```
  clock_gettime(CLOCK_REALTIME, …);
  func((void*)&first_elem);
  func((void*)&bad_elem);
  clock_gettime(CLOCK_REALTIME, …);

  clock_gettime(CLOCK_REALTIME, …);
  pthread_create(&thread_1, NULL,func, (void*)&first_elem);
  pthread_create(&thread_2, NULL,func, (void*)&bad_elem);
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  clock_gettime(CLOCK_REALTIME, …);

  clock_gettime(CLOCK_REALTIME, …);
  pthread_create(&thread_1, NULL,func, (void*)&first_elem);
  pthread_create(&thread_2, NULL,func, (void*)&good_elem);
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  clock_gettime(CLOCK_REALTIME, …);
}
```

https://github.com/MJjainam/falseSharing/

Impact of False Sharing

```
int array[100];

void *func(void *param) {
  int   index = *((int*)param);
  int   i;
  for (i = 0; i < 100000000; i++)
    ar
}

int ma
  int
  int
  int

  pthread_t thread_1;
  pthread_t thread_2;
```

```
  clock_gettime(CLOCK_REALTIME, …);
  func((void*)&first_elem);
  func((void*)&bad_elem);
  clock_gettime(CLOCK_REALTIME, …);

  clock_gettime(CLOCK_REALTIME, …);
                                        first_elem);
                                        bad_elem);
                                        first_elem);
                                        good_elem);
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  clock_gettime(CLOCK_REALTIME, …);
}
```

	Millisecond
Sequential computation	351
With false sharing	465
Without false sharing	168

https://github.com/MJjainam/falseSharing/

False Sharing in Real Applications

- Issues reported in Linux kernel, JVM, Boost, …

Fixing false sharing improved a metric of interest by almost 3X

False Sharing is Everywhere

```
// Global variables
me = 1;
you = 2;
```

```
// Class/struct fields
class X {
  int me;
  float you;
};
```

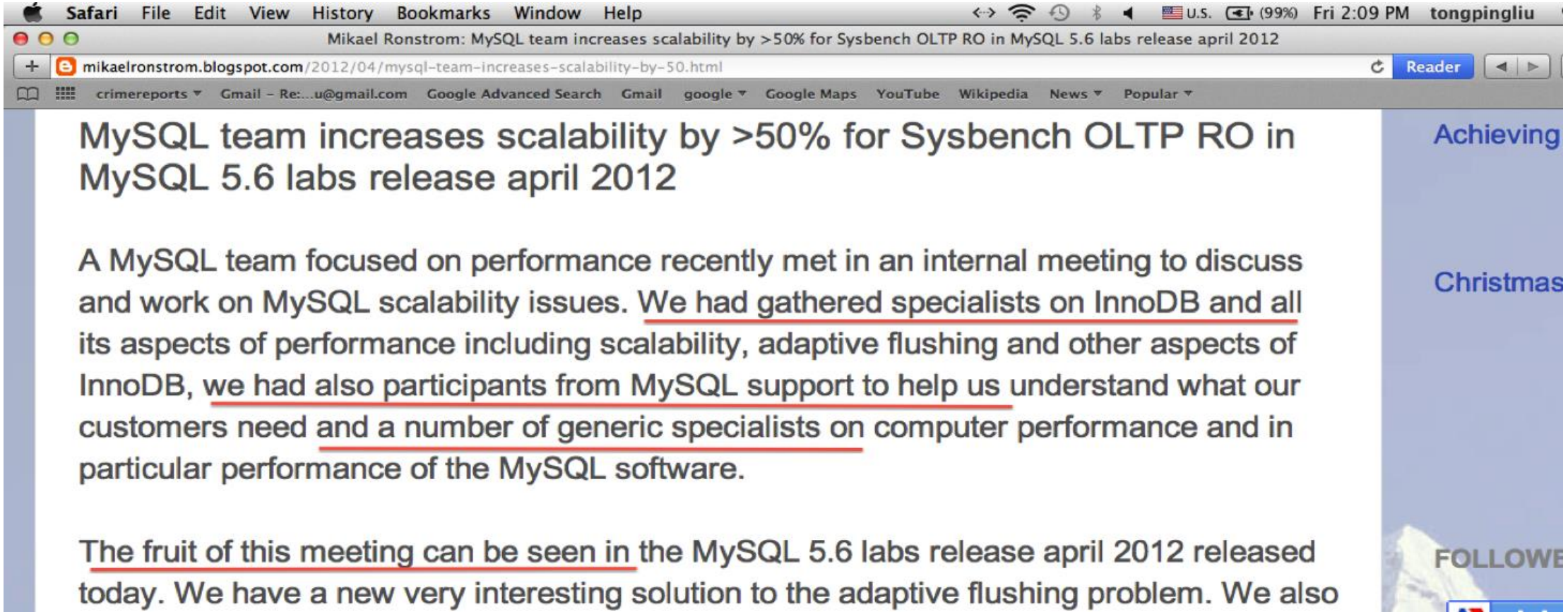
```
// Heap objects
me = new Foo();
you = new Bar();
```

```
// Array accesses
array[me] = 12;
array[you] = 13;
```

False Sharing Mitigation Techniques

- Compiler optimizations (cache block padding)
- Cache conscious programming
- Coherence at load/store granularity?

Fixing False Sharing is Non-trivial



Fixing False Sharing is Non-trivial

- Problem is often embedded inside the source code
- Sensitive to
  - Object placements on the cache line
  - Memory allocation sequence or memory allocator
  - Hardware platform with different cache line sizes

gcc accidentally eliminates false sharing in Phoenix linear_regression benchmark at certain optimization levels, while LLVM does not do so at any optimization level

Related Work on False Sharing

Sheriff	Liu and Berger, OOPSLA'11	detect and repair unmanaged languages
Plastic	Nanavati et al., EuroSys'13	
Laser	Luo et al, HPCA'16	
Cheetah	Liu and Liu, CGO'16	detection only
Predator	Liu et al., PPoPP'14	
Intel vTune Amplifier XE		
Oracle Java 8 @Contended		annotation and repair
REMIX	Eizenberg et al., PLDI'16	detect and repair in managed runtimes
TMI	DeLoizer et al., MICRO'17	

False Sharing Problem in JVMs

- JVMs provide automatic layout of class fields at load time
  - Sort fields by descending order of size
  - Pack reference fields to help GC process a contiguous pack of reference fields
  - Padding as in C/C++ may not work in Java since the JVM can remove or reorder unused fields
  - Copying GCs move around objects
- Single-threaded environment
  - Fields accessed together in time should be nearby in space
- Multithreaded environment
  - Not so straightforward, cannot just aim to reduce capacity misses

---

https://blogs.oracle.com/dave/java-contended-annotation-to-help-reduce-false-sharing

Easy Thing First! Java 8 @Contended

- Now that you know about false sharing, use @sun.misc.Contended in Java to (hopefully) get benefits for free

- @Contended helps avoid false sharing, but does not automatically detect sources of contention

---

https://blogs.oracle.com/dave/java-contended-annotation-to-help-reduce-false-sharing

Easy Thing First! Java 8 @Contended

```
@Contended
public static class ContendedTest2 {
  private Object plainField1;
  private Object plainField2;
  private Object plainField3;
  private Object plainField4;
}

$ContendedTest2: field layout
Entire class is marked contended
@140 --- instance fields start ---
@140 "plainField1" Ljava.lang.Object;
@144 "plainField2" Ljava.lang.Object;
@148 "plainField3" Ljava.lang.Object;
@152 "plainField4" Ljava.lang.Object;
@288 --- instance fields end ---
@288 --- instance ends ---
```

```
public static class ContendedTest1 {
@Contended
  private Object contendedField1;
  private Object plainField1;
  private Object plainField2;
  private Object plainField3;
  private Object plainField4;
}

$ContendedTest1: field layout
@ 12 --- instance fields start ---
@ 12 "plainField1" Ljava.lang.Object;
@ 16 "plainField2" Ljava.lang.Object;
@ 20 "plainField3" Ljava.lang.Object;
@ 24 "plainField4" Ljava.lang.Object;
@156 "contendedField1" Ljava.lang.Object; (contended, group
= 0)
@288 --- instance fields end ---
@288 --- instance ends ---
```

http://beautynbits.blogspot.com/2012/11/the-end-for-false-sharing-in-java.html

Easy Thing First! Java 8 @Contended

```
public static class ContendedTest4 {
  @Contended
  private Object contendedField1;
  @Contended
  private Object contendedField2;
  private Object plainField3;
  private Object plainField4;
}
```

```
$ContendedTest4: field layout
@ 12 --- instance fields start ---
@ 12 "plainField3" Ljava.lang.Object;
@ 16 "plainField4" Ljava.lang.Object;
@148 "contendedField1" Ljava.lang.Object;
(contended, group = 0)
@280 "contendedField2" Ljava.lang.Object;
(contended, group = 0)
@416 --- instance fields end ---
@416 --- instance ends ---
```

http://beautynbits.blogspot.com/2012/11/the-end-for-false-sharing-in-java.html

Sheriff: Precise Detection and Automatic Mitigation

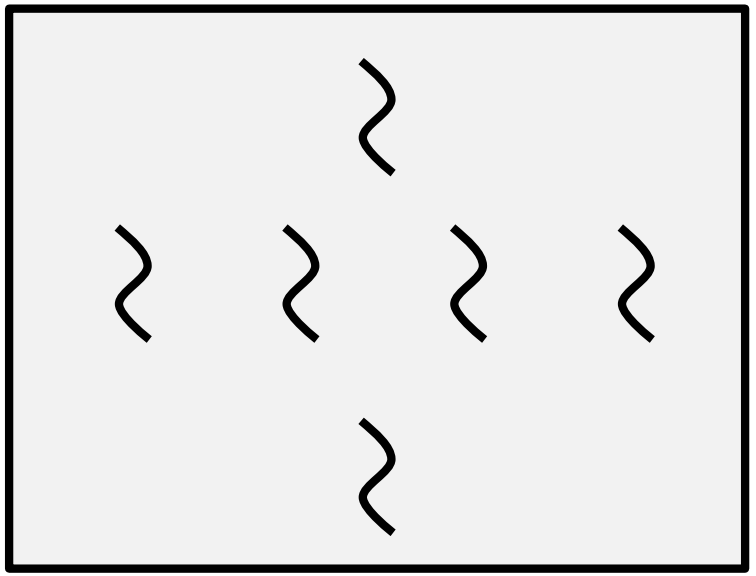
- Sheriff is a software-only solution that provides
  - Per-thread memory protection – allows each thread to track memory accesses independently of other thread's accesses
  - Memory isolation – allows each thread to read from and write to memory without interference from other threads

---

T. Liu and E. Berger. Sheriff: Precise Detection and Automatic Mitigation of False Sharing. OOPSLA 2011.

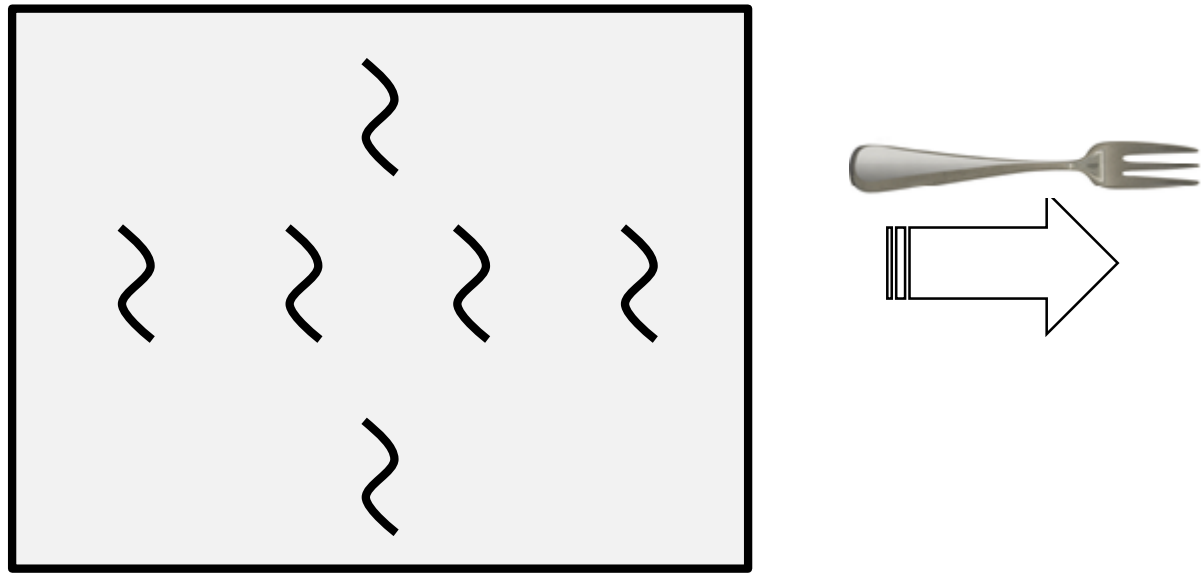
Isolated Memory Access

shared address space

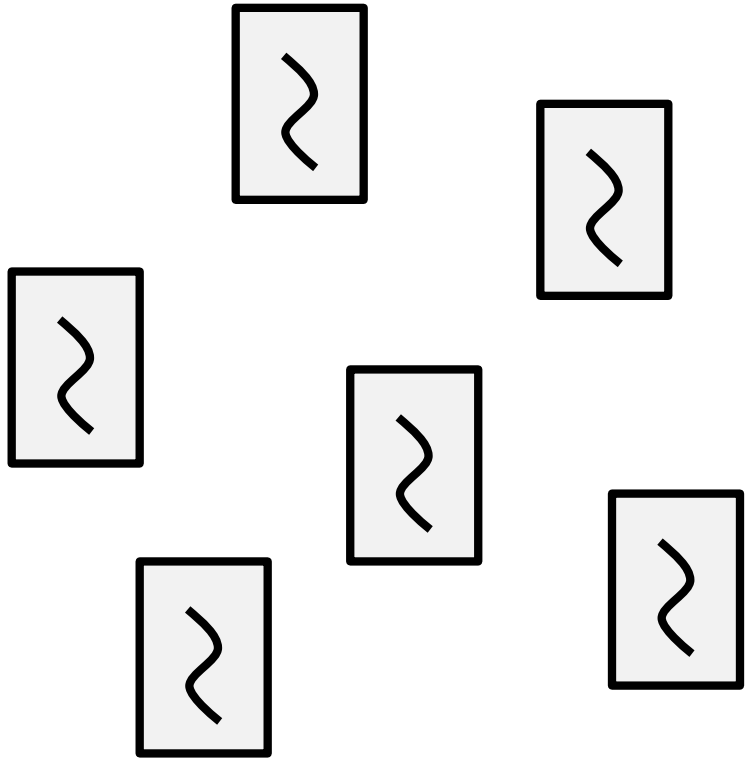


Isolated Memory Access

shared address space



disjoint address space



Tradeoff in Faking Threads with Processes

- Processes are mapped to different CPUs, while threads are mapped to the same CPU to maximize locality
- Using processes allows Sheriff to use
  - Per-thread page protection to detect false conflicts
  - Isolates thread's memory from other threads which implies thread's do not write to each other's cache lines

Isolated Memory Accesses

- Processes have separate address spaces $\Rightarrow$ Implies that updates to shared memory are not visible
- Challenges
  - Sheriff now needs to explicitly manage shared resources like file descriptors
  - Uses memory mapped files to share shared data (e.g., globals, heap) across processes
    - Two copies are created – one is read-only and the other (CoW) is for local updates
    - Private mapping initially points to the read-only page

Shared Memory Updates

Updates are made visible only at synchronization points

Pthreads

```
Lock();

XXX;

Unlock();

YYY;

Lock();
```

Sheriff

```
Begin_isolated_execution
XXX;    //isolated execution
Commit_local_changes
Begin_isolated_execution
YYY;    //isolated execution
Commit_local_changes
```

Sheriff in Action!

Initialization

- Create shared and local mappings for heap and global variables

Transaction begin

- Write protect shared pages, future writes will trap

Execution

- Records pages with faulted addresses and unprotects the page
- Creates a twin page for diffing before a page is modified
- Performs CoW to create a private page

Transaction end

- Commits only diffs between the twin and the private pages

Sheriff-Detect: Detect False Sharing

- Idea
  - Any cache line with different contents in the private page and the twin page is due to false sharing
  - Can have high overhead for pages that are unshared
- Insight
  - For false sharing, two threads must simultaneously access the page containing the cache line $\Rightarrow$ Implies the page must be shared
  - Sheriff-Detect keeps track of the number of writers to a shared page
  - Problem if there is a cache line with one writer and rest are readers

Sheriff-Protect: Runtime to Avoid False Sharing

- Sheriff-Detect may not work satisfactorily
  - Padding may degrade performance due to cache effects and increased memory consumption
  - Source code may not be available to fix false sharing issues
- Insight – Delay updates to avoid false sharing
- Protects only small objects
  - Benefit of protection is greater than large objects like arrays
  - Cost of protection via committing updates is going to be lower

Drawbacks of Sheriff

Cannot detect read-write false sharing

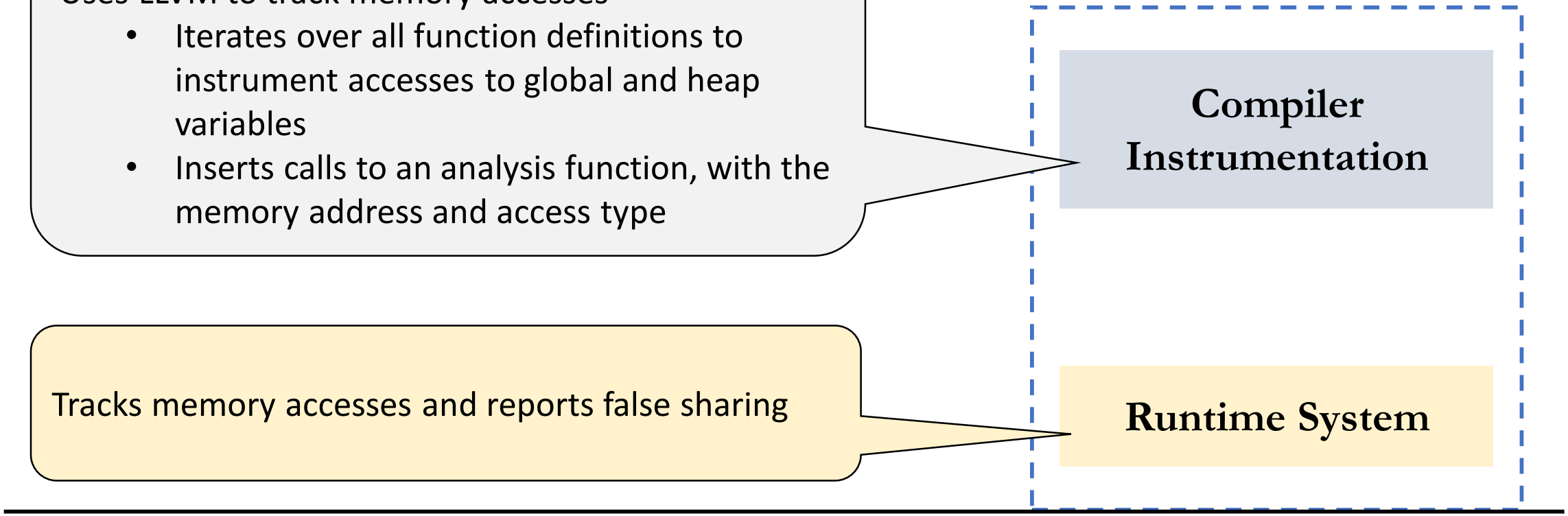
Can only detect false sharing in the observed executions

Predator: Predictive False Sharing Detection

Uses LLVM to track memory accesses

- Iterates over all function definitions to instrument accesses to global and heap variables
- Inserts calls to an analysis function, with the memory address and access type

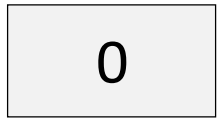
Tracks memory accesses and reports false sharing



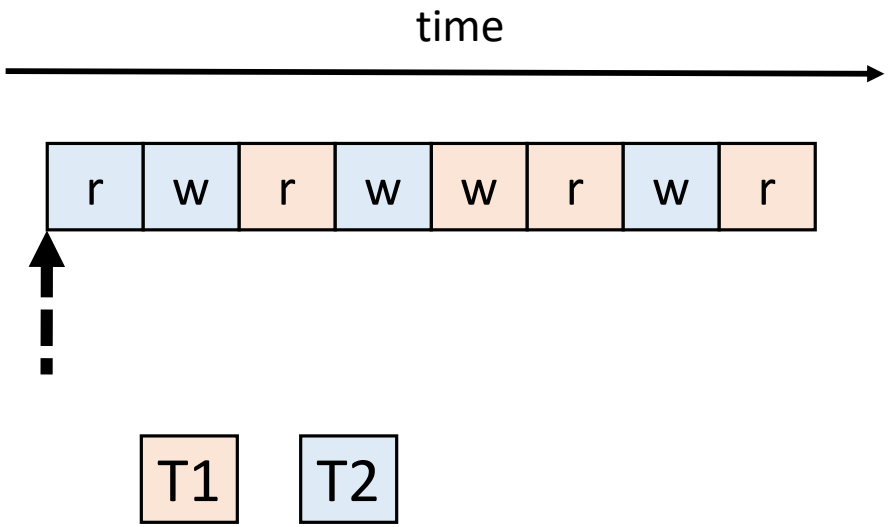
T. Liu et al. Predator: Predictive False Sharing Detection. PPoPP 2014.

Track Cache Invalidations

Each Entry: { Thread ID, Access Type}



# of invalidations



Rules for Cache History Table

- For each read R,
  - If Table T is full, no need to record R
  - If T is not full and existing entry has a different thread ID, then record R
- For each write W,
  - If T is full, then W can cause a cache invalidation since at least one of two existing entries has a different thread ID. Record invalidation. Update the existing entry.
  - If T is not full, check whether W and the existing entry have the same thread ID
    - Same thread ID – W cannot cause a cache invalidation, update existing entry with W
    - Different thread ID – Record an invalidation on this line caused by W. Record this invalidation, and update the existing entry with W.

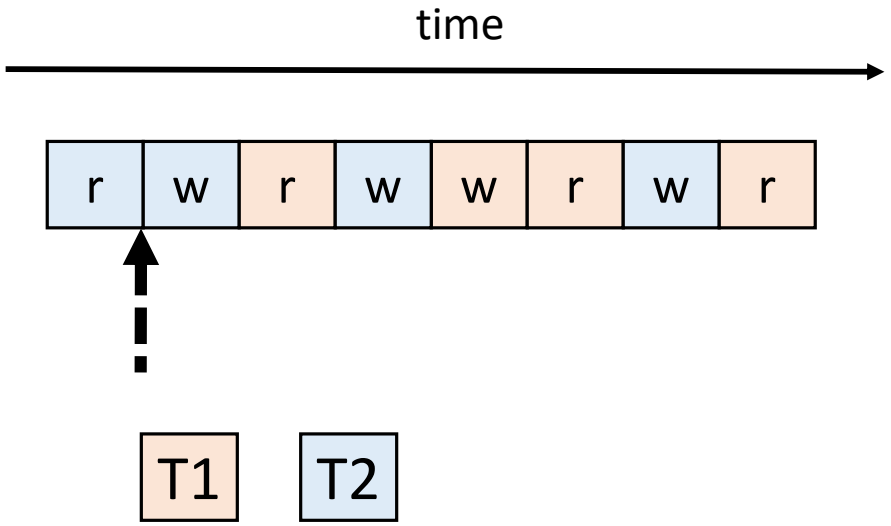
Track Cache Invalidations

Each Entry: { Thread ID, Access Type}

T2	r	0	0

0

# of invalidations



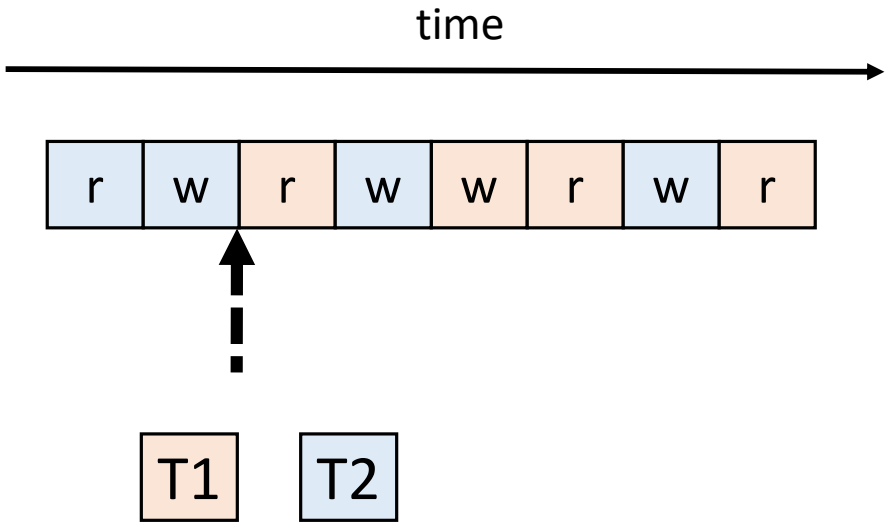
Track Cache Invalidations

Each Entry: { Thread ID, Access Type}

T2	w	0	0

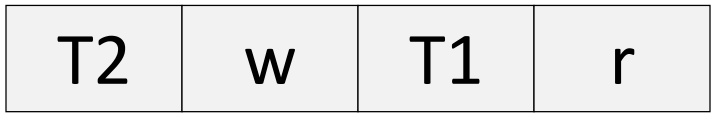
0

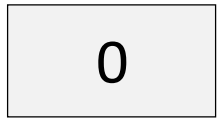
# of invalidations



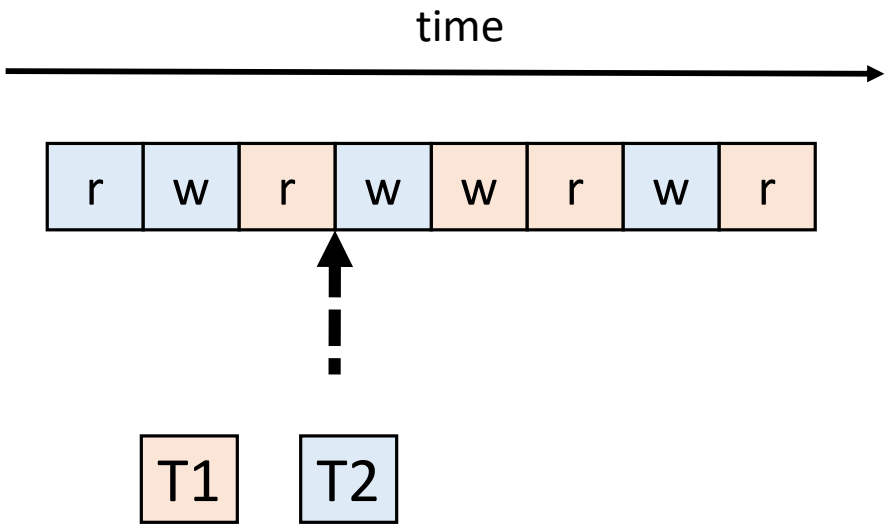
Track Cache Invalidations

Each Entry: { Thread ID, Access Type}





# of invalidations



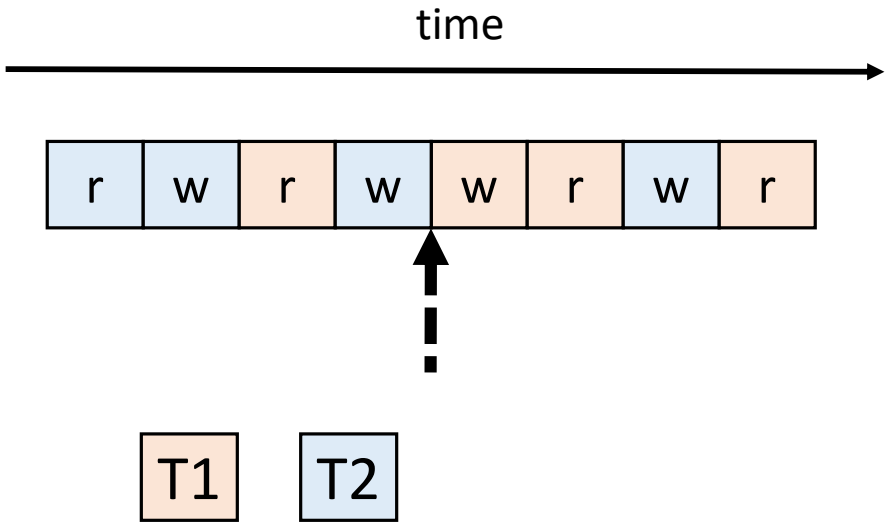
Track Cache Invalidations

Each Entry: { Thread ID, Access Type}

T2	w	0	0

1

# of invalidations



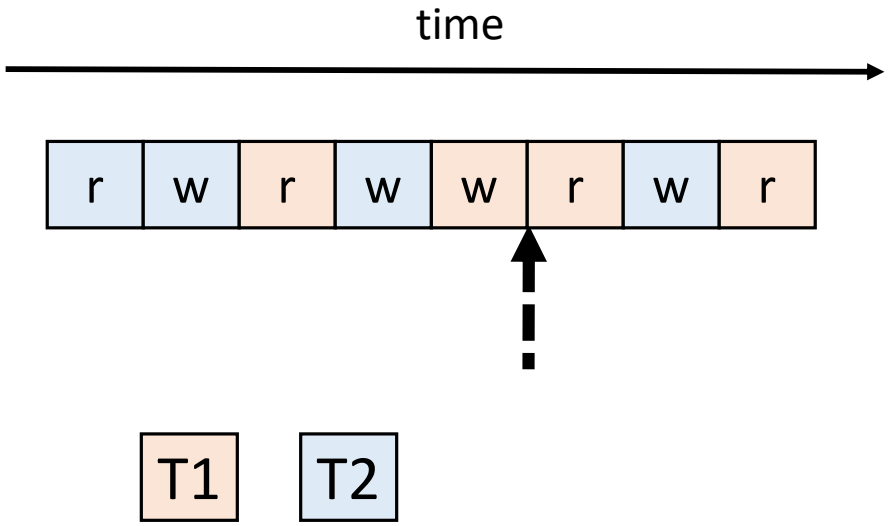
Track Cache Invalidations

Each Entry: { Thread ID, Access Type}

T1	w	0	0

2

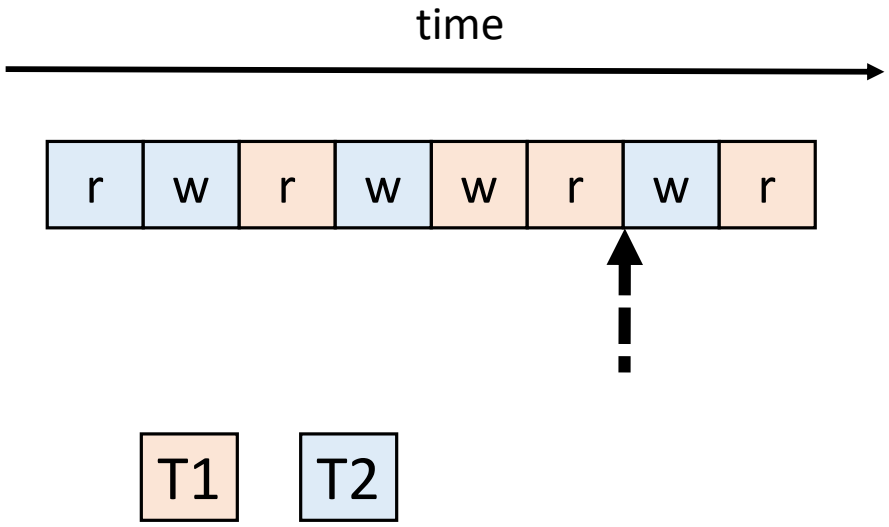
# of invalidations



Track Cache Invalidations

Each Entry: { Thread ID, Access Type}

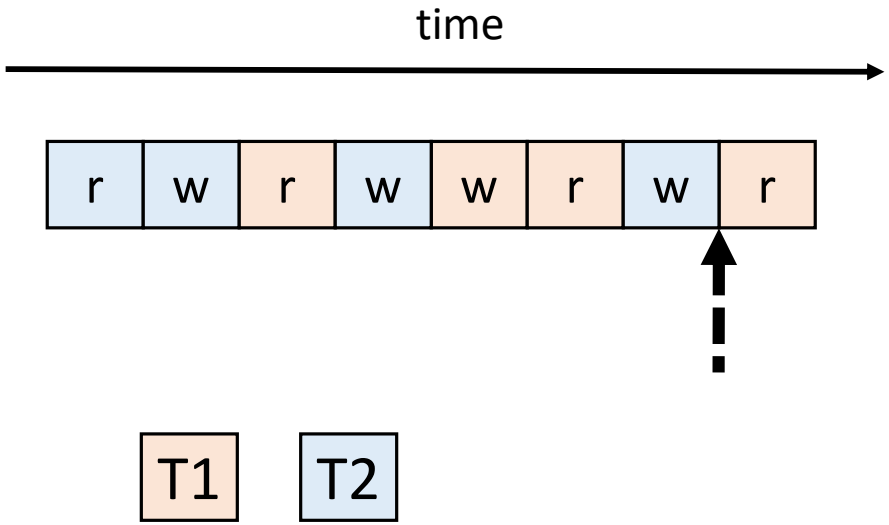
# of invalidations



Track Cache Invalidations

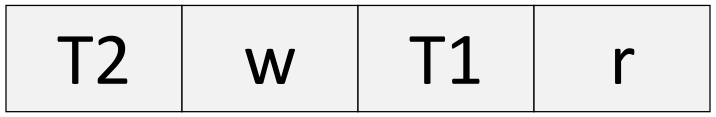
Each Entry: { Thread ID, Access Type}

# of invalidations

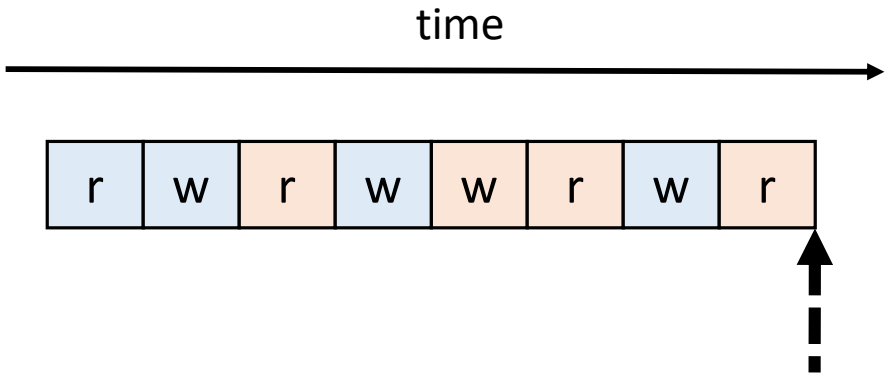


Track Cache Invalidations

Each Entry: { Thread ID, Access Type}



# of invalidations

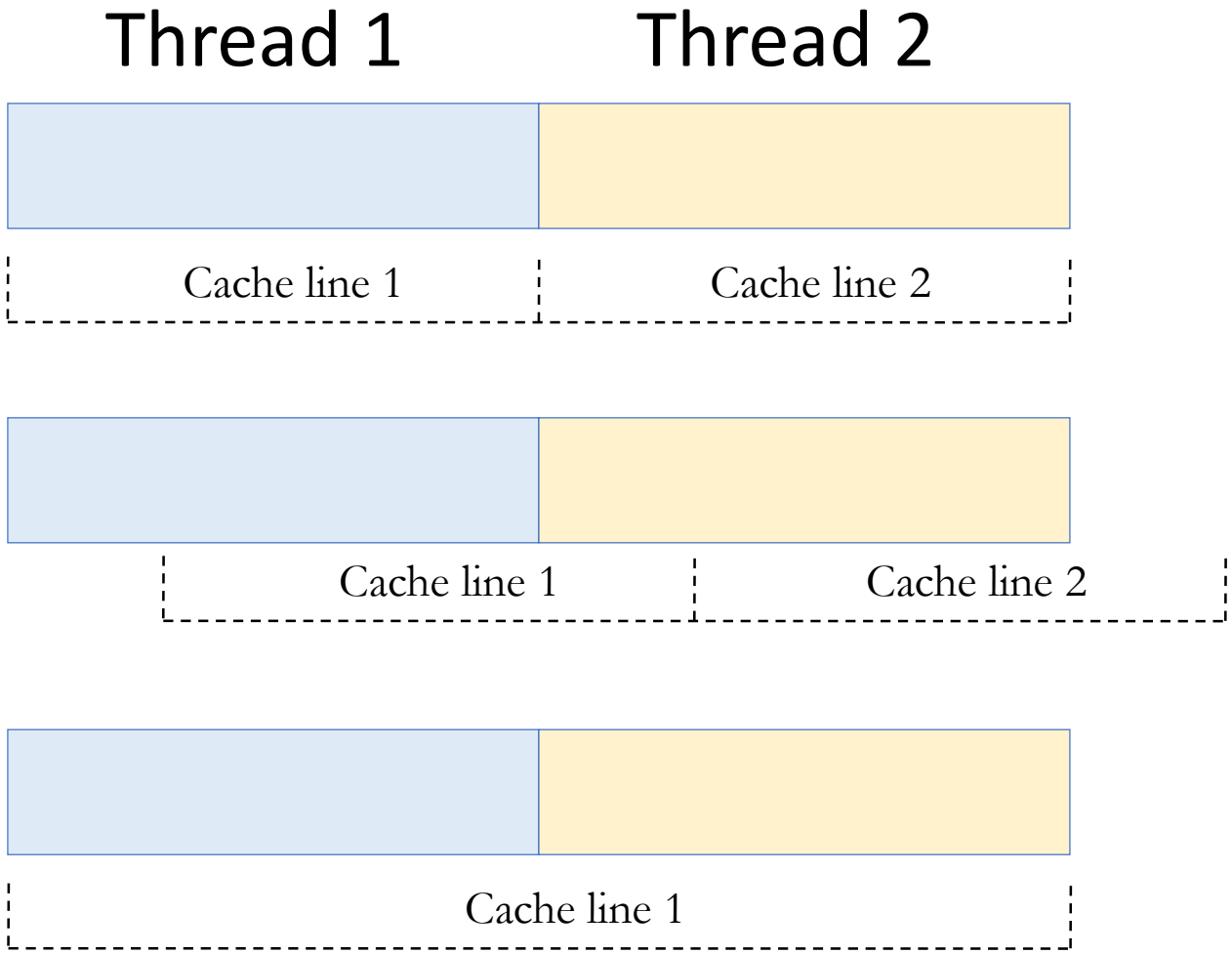


Is that it?

- Well, true sharing also leads to cache invalidations
- Predator maintains precise per-cache-line-offset metadata

Why do we need to predict false sharing?

- Object alignment impacts the occurrence of false sharing



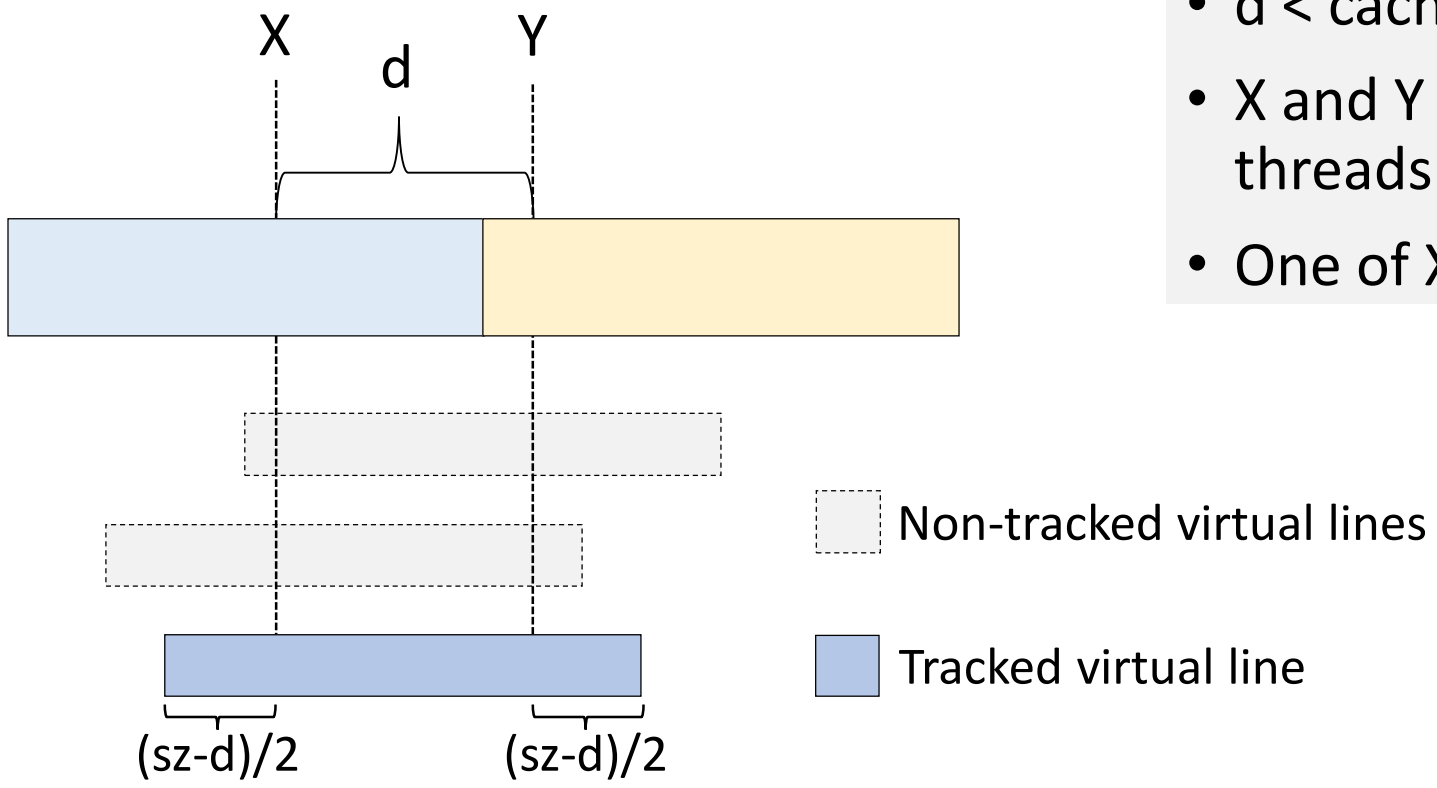
Impact on Object Alignment

- 32-bit platform $\Rightarrow$ 64-bit platform
- Different memory allocator
- Different compiler or optimizations
- Different allocation order by changing the code
- Run on hardware with different cache line size

Prediction in Predator

- Insight
  - Only accesses to adjacent lines can lead to potential false sharing
- Virtual cache line
  - Contiguous memory range spanning multiple physical cache lines
  - Starting address need not be a multiple of the cache line size
  - 64-byte line can range from [0, 64) or [8, 72) bytes
- Find “hot” access offsets X and Y
  - X in cache line L, and Y in adjacent cache line, and both X and Y are in the same virtual cache line
  - At least one of X and Y is a write
  - X and Y are accessed by different threads

Track Invalidations on Virtual Cache Lines



- d < cache line size (sz)
- X and Y are accesses from different threads
- One of X and Y accesses is a write