

# CS 636: Performance Analysis of Concurrent Programs

Swarnendu Biswas

Semester 2020-2021-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

# 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

# Functional and Performance Bugs

## Functional Bugs

- Well-defined notion of success and failure
- Correctness requirements usually do not change over time
  - Other than significant changes in the specification
- More focus on researched testing methodologies
- Rate of bugs generally flatten out with maturity

## Performance Bugs

- Difficult to detect because of no failure symptoms
- Performance requirements may evolve over time
- Relative lack of formalized testing methodologies
- Rate of bugs reported have no trends

# 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 volatile boolean initialized;  
    synchronized void initialize() {  
        if (!this.initialized)  
            this.setInitialized(true);  
    }  
    void setInitialized(boolean b) { this.initialized = b; }  
    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;  
    }  
}
```

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 in Java? Would volatile not suffice?

```
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

# 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**

# Example 2

Mozilla developers implemented a procedure `nsImage::Draw` for figure scaling, compositing, and rendering, which is a waste for transparent figures. This problem did not catch developers' attention until two years later when 1px\*1px 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*.

## 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 GC process 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.

# 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. Debugging 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.

# 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.



# 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 (bookmark all (tabs))

Wrong API interpretation

Redundant work (MySQL `fastmutex_lock`)

- Wrong functional implementation

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

- Many synchronization fixes are just because of performance reasons

Cross core/node data communication

Miscellaneous

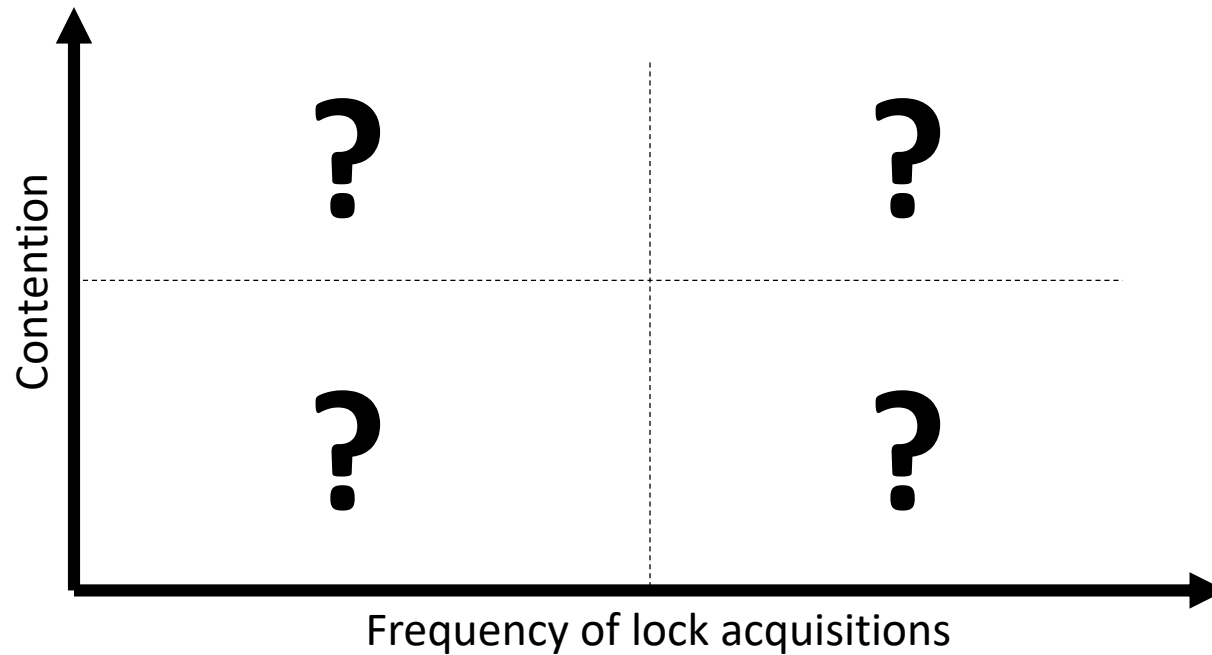
- Poor data structure choices, design/algorithm issues, data partitioning, load balancing and task stealing

# Dealing with Performance Bugs

- Compilers and hardware optimizations may not always fix performance problems
- Automation support is limited and is still being explored
  - Current strategies involve random testing and feedback of testers
  - Design better performance tests
  - Use annotation systems
  - ...

# Tracking Synchronization Bottlenecks

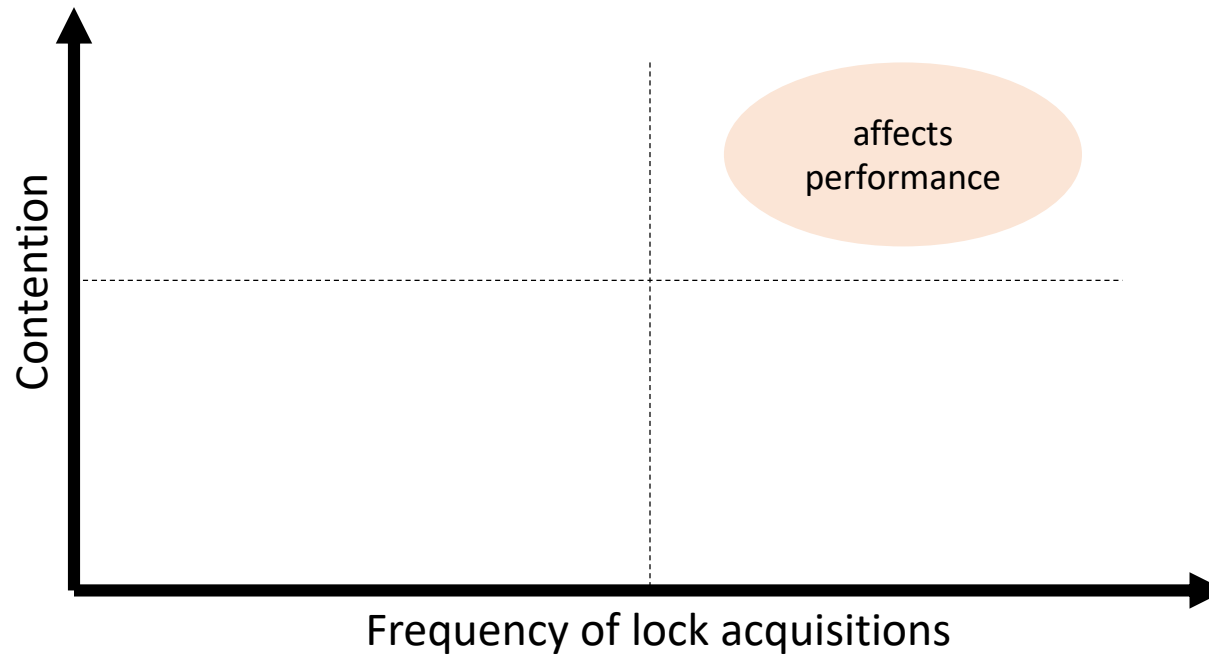
# Synchronization-Related Factors That Affect Performance



---

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

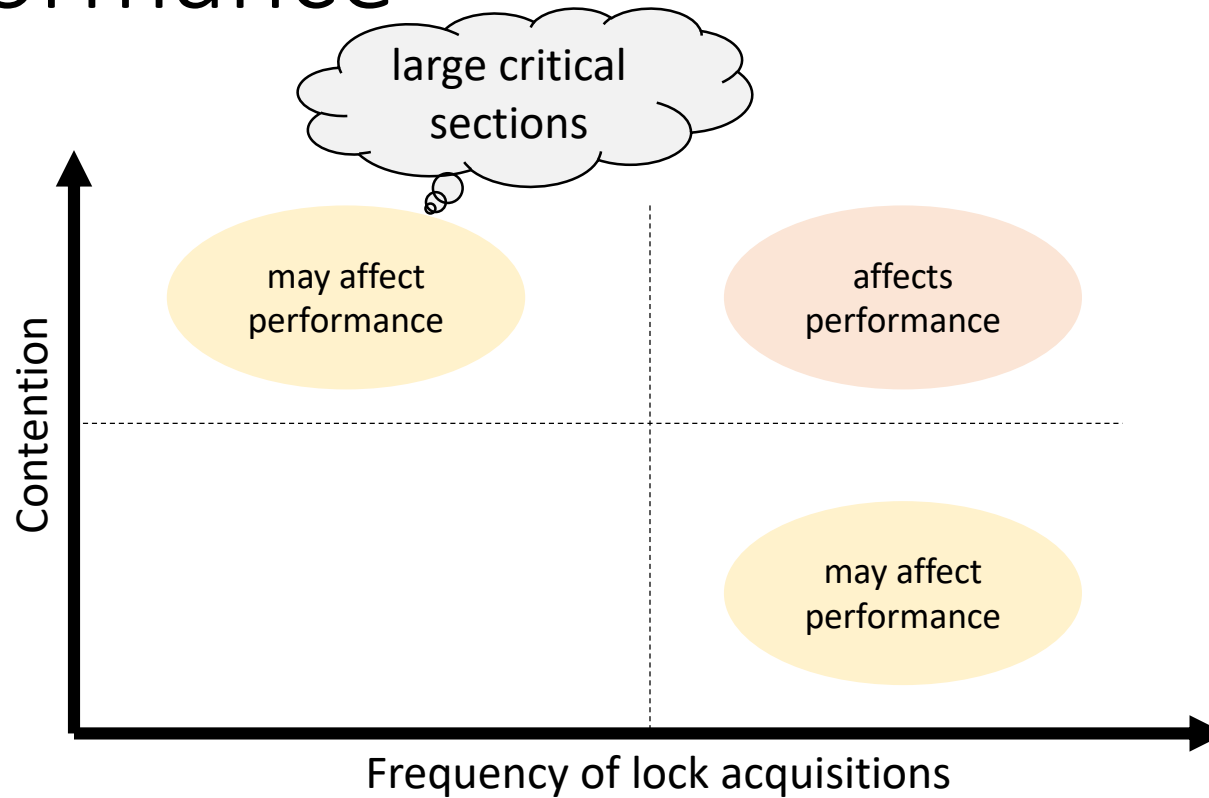
# Synchronization-Related Factors That Affect Performance



---

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.

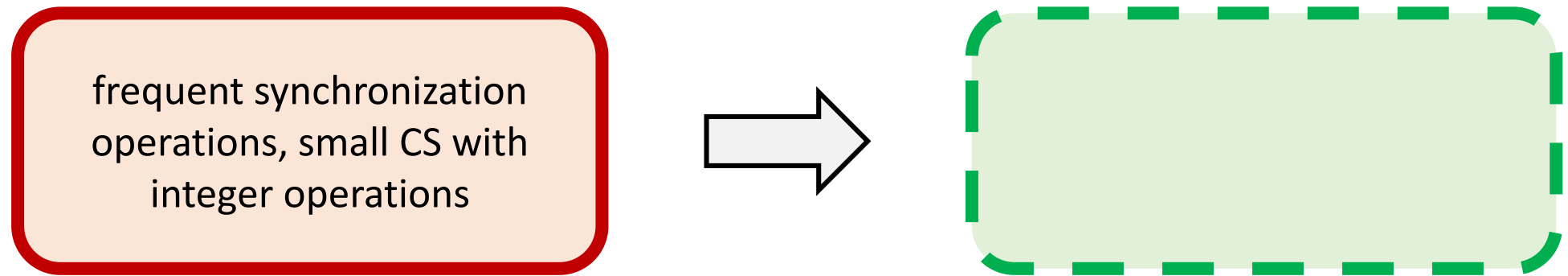
# Spectrum of Synchronization Operations

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



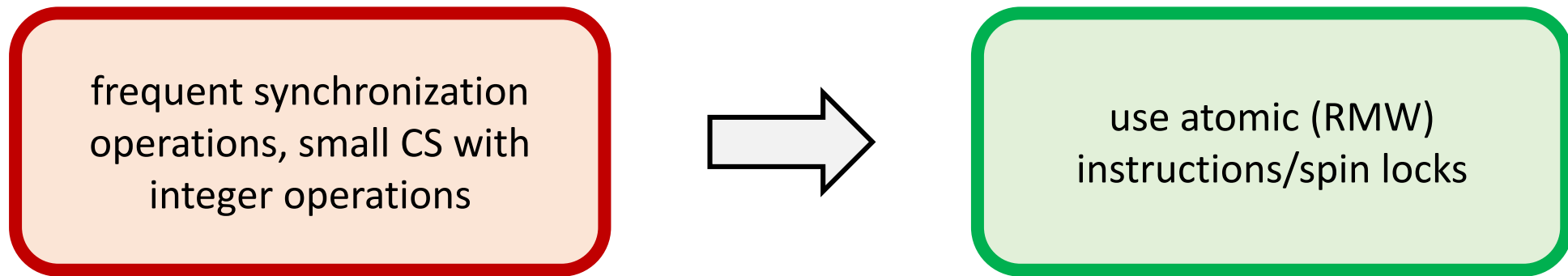
# Reasons for Synchronization-Related Performance Loss

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



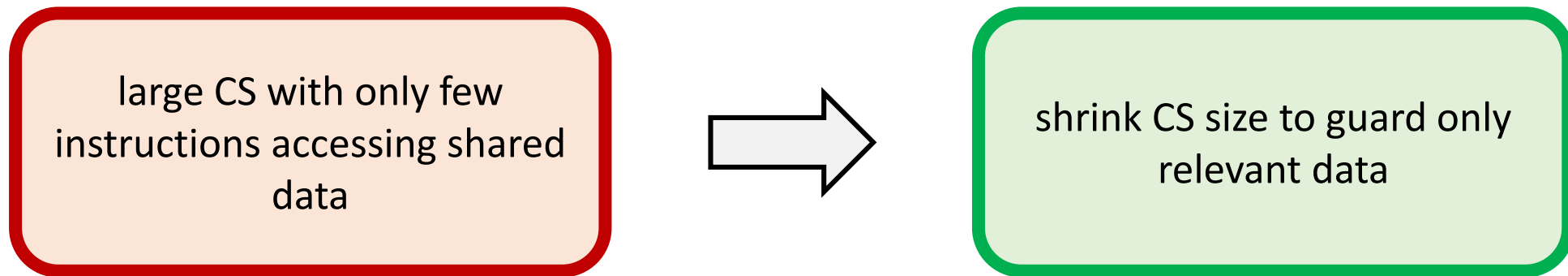
# Reasons for Synchronization-Related Performance Loss

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



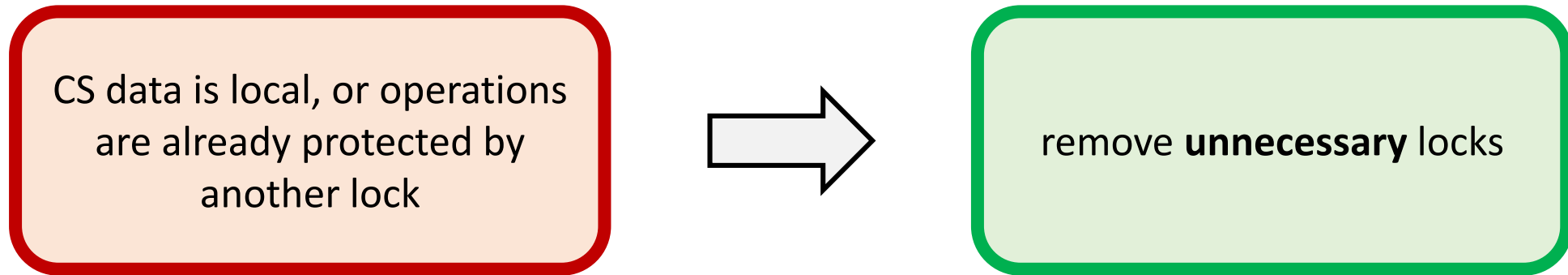
# Reasons for Synchronization-Related Performance Loss

- 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.



# Potential Parallelism Hurt by Synchronization

## Thread 1

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)
```

## Thread 2

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(Y)
if (!var)
    hash_tbl.add(Y);
UNLOCK(hash_tbl.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 cache 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

- Track memory accesses in critical sections, detect conflicts, and perform rollbacks
- “Best effort” – can fallback to acquire the lock and reexecute non-speculatively
- Need to identify opportunities for lock elision

## TM

- Track memory accesses in transactions, detect conflicts, and perform rollbacks
- TM generally is always speculative
- Complete program execution can be transactional

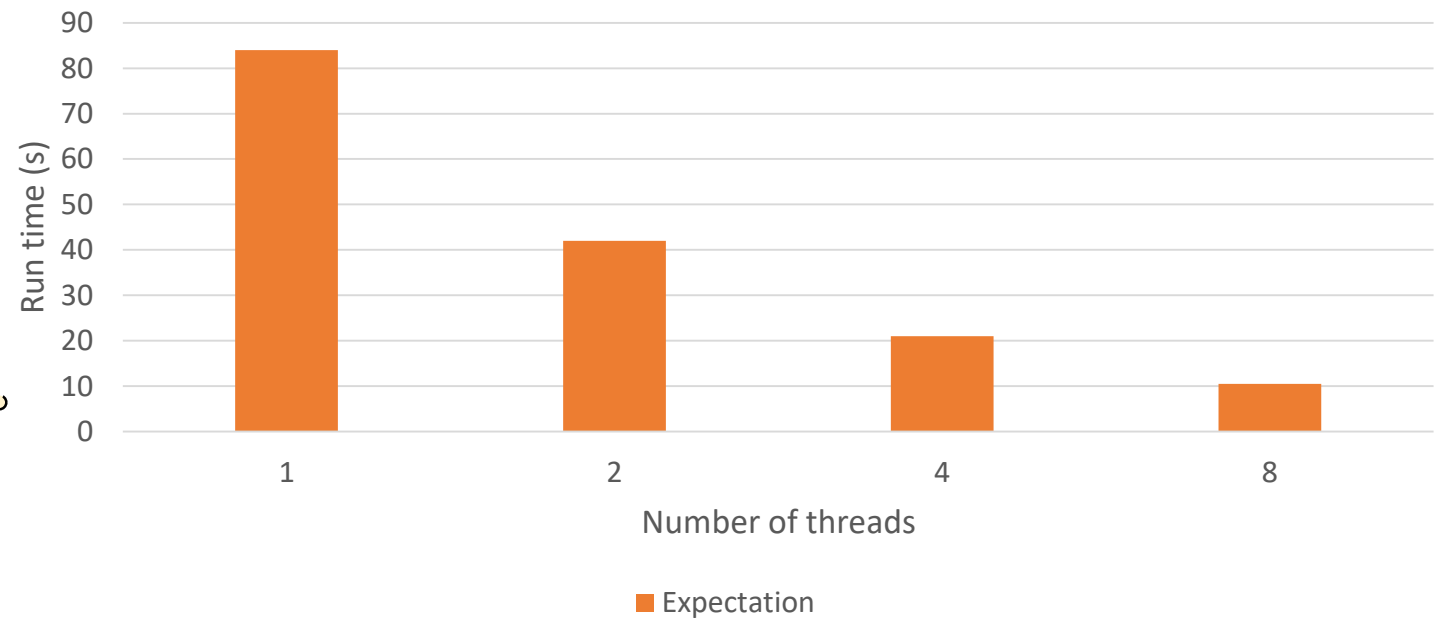
# Dealing with False Sharing

# Multicore Parallelism is Easy

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

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

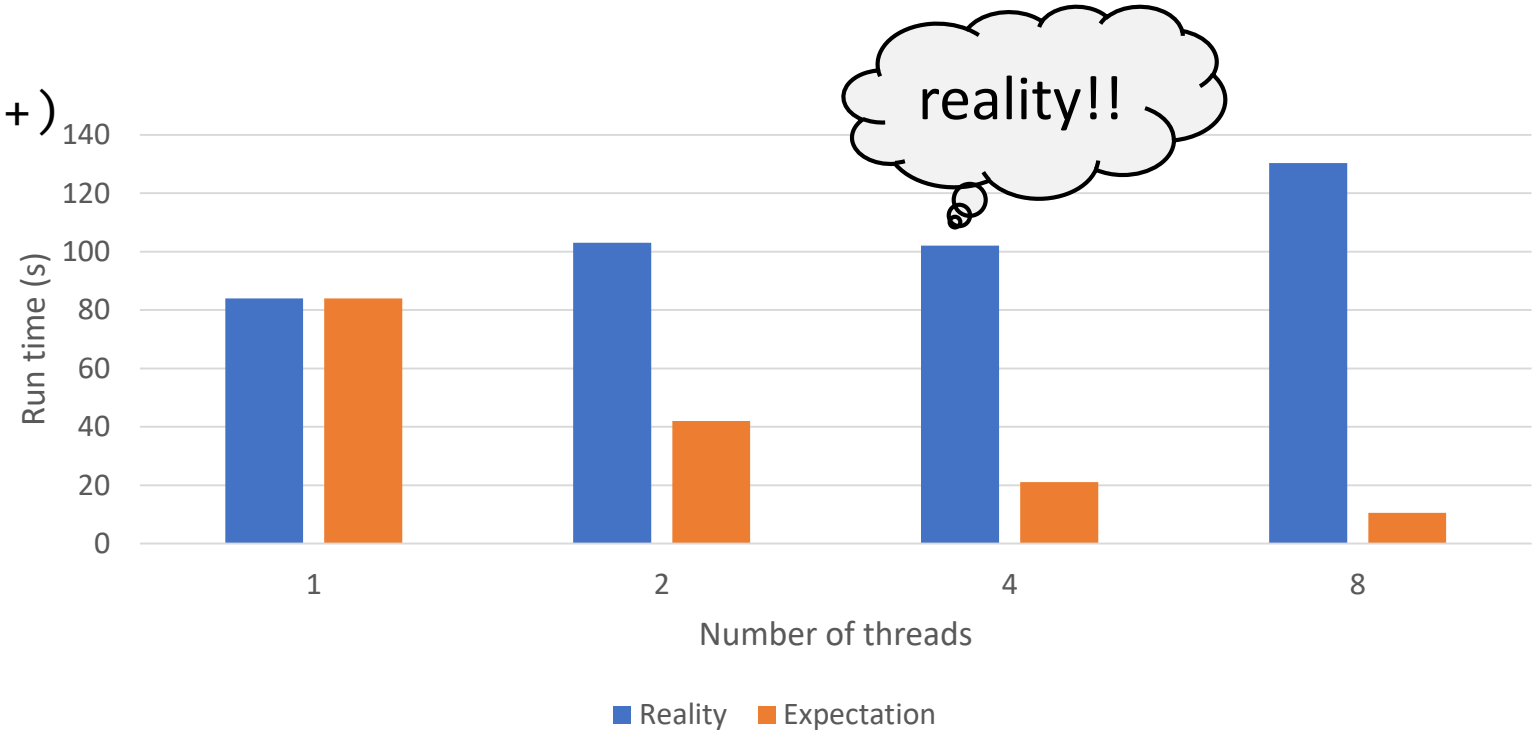
We are expecting  
strong scaling



# Multicore Parallelism is Easy

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

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

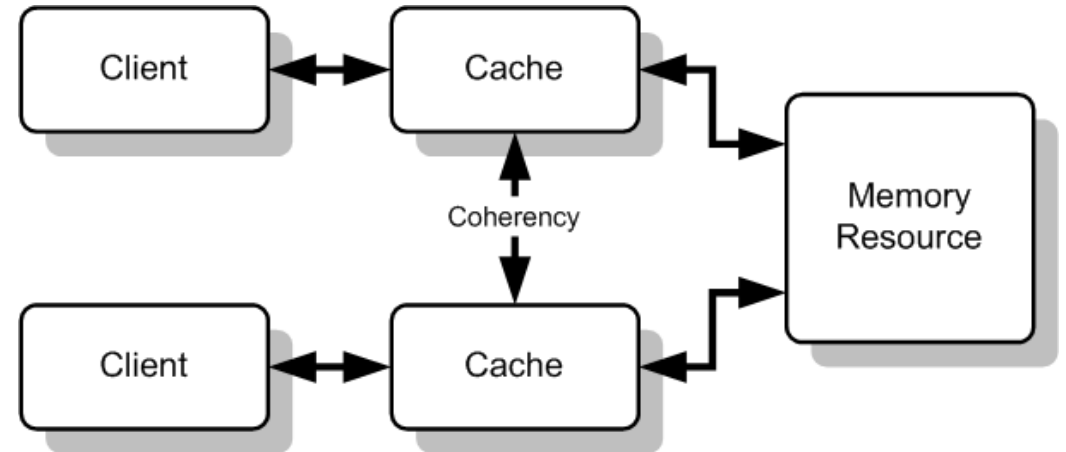


# Shared Memory Multiprocessors

- Processors employ private caching of data to improve performance
  - Private copies of shared data must be “coherent”
  - Roughly speaking, all copies must have the same value (enough if this holds eventually)
- For sequential programs, a memory location must return the latest value written to it
- For parallel programs, we expect the same provided “latest” is well-defined
  - For now, latest value of a location is the latest value “committed” by any thread/process

# Cache Coherence

- Multicore processors implement a cache coherence protocol to keep private caches in sync
- A cache coherence protocol is a set of actions that ensure that a load to address A returns the “last committed” value to A
  - Operates on whole cache lines (usually 64 bytes)



# Need for Coherence: Example 1

- Assume 3 cores with write-through caches
  - C0: reads x from memory, puts it in its cache, and gets the value 5
  - C1: reads x from memory, puts it in its cache, and gets the value 5
  - C1: writes x=7, updates its cached value and memory value
  - C0: reads x from its cache and gets the value 5
  - C2: reads x from memory, puts it in its cache, and gets the value 7 (now the system is completely incoherent)
  - C2: writes x=10, updates its cached value and memory value
- 

M. Chaudhuri. Cache Coherence. CASS 2018.

# Need for Coherence: Example 2

- Assume 3 cores with write-back caches
- C0: reads x from memory, puts it in its cache, and gets the value 5
- C1: reads x from memory, puts it in its cache, and gets the value 5
- C1: writes x=7, updates its cached value
- C0: reads x from its cache and gets the value 5
- C2: reads x from memory, puts it in its cache, and gets the value 7
- C2: writes x=10, updates its cached value



# Need for Coherence: Example 2

- The lines in C1 and C2 are dirty, while the line is clean in C0
- Eviction of the lines from C1 and C2 will write the data back
- We will lose a store depending on the order of writebacks
  - Suppose C2 evicts the line first, and then C1
  - Final memory value is 7: we lost the store  $x=10$  from C2

# What went wrong?

- For write-through cache
  - The memory value may be correct if the writes are correctly ordered
  - But the system allowed a store to proceed when there is already a cached copy
  - **Lesson learned:** must invalidate all cached copies before allowing a store to proceed
- For writeback cache
  - Problem is even more complicated: stores are no longer visible to memory immediately
  - Writeback order is important
  - **Lesson learned:** do not allow more than one copy of a cache line in dirty state

# Solutions

- Must invalidate all cached copies before allowing a store to proceed
  - Need to know where the cached copies are
- Solution1: Just tell everyone that you are going to do a store
  - Leads to broadcast snoopy protocols
  - Popular with small-scale machines
  - Typically, the interconnect is a shared bus
- Solution2: Keep track of the sharers and invalidate them when needed
  - Where and how is this information stored?
  - Leads to directory-based scalable protocols

# Solutions

- Directory-based protocols
  - Maintain one directory entry per memory block
  - Each directory entry contains a sharer bitvector and state bits
- Do not allow more than one copy of a cache line in dirty state
  - Need some form of access control mechanism
  - Before a processor does a store it must take “permission” from the current “owner” (if any)
  - Need to know who the current owner is: either a processor or main memory
  - Earlier solutions apply here also: broadcast to everybody or request the owner

---

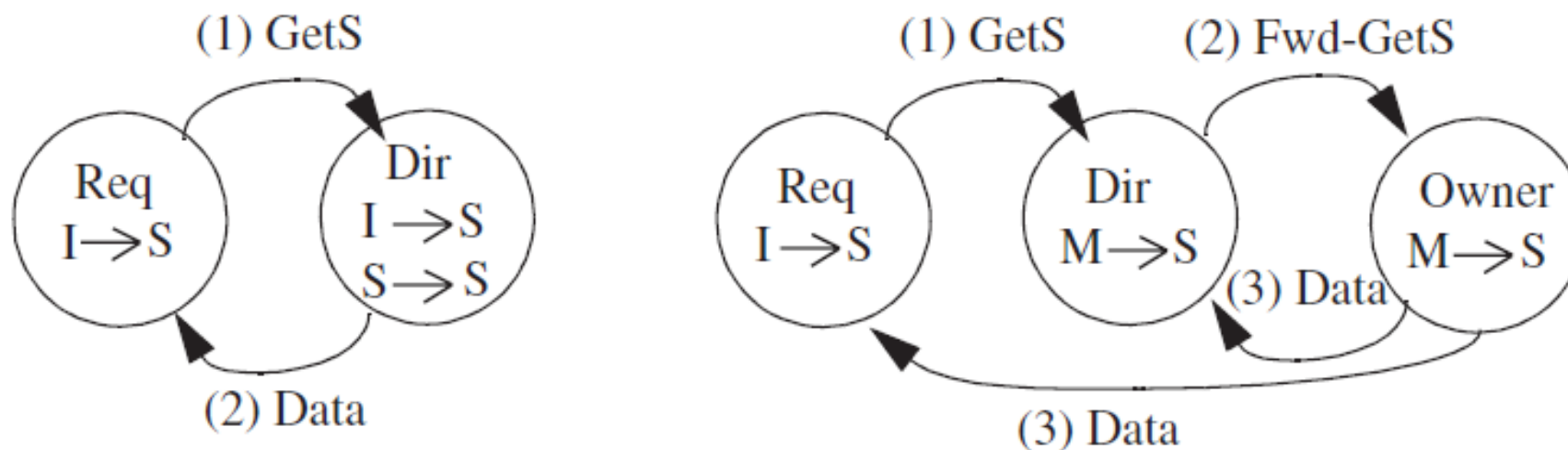
M. Chaudhuri. Cache Coherence. CASS 2018.

# Types of Coherence Protocols

- Two main classes of protocols: dictates what action should be taken on a store
  - **Invalidation-based** protocols invalidate sharers when a store miss appears
  - **Update-based** protocols update the sharer caches with new value on a store
  - Advantage of update-based protocols: sharers continue to hit in the cache while in invalidation-based protocols sharers will miss next time they try to access the line
  - Advantage of invalidation-based protocols: only store misses go on bus and subsequent stores to the same line are cache hits
- 

M. Chaudhuri. Cache Coherence. CASS 2018.

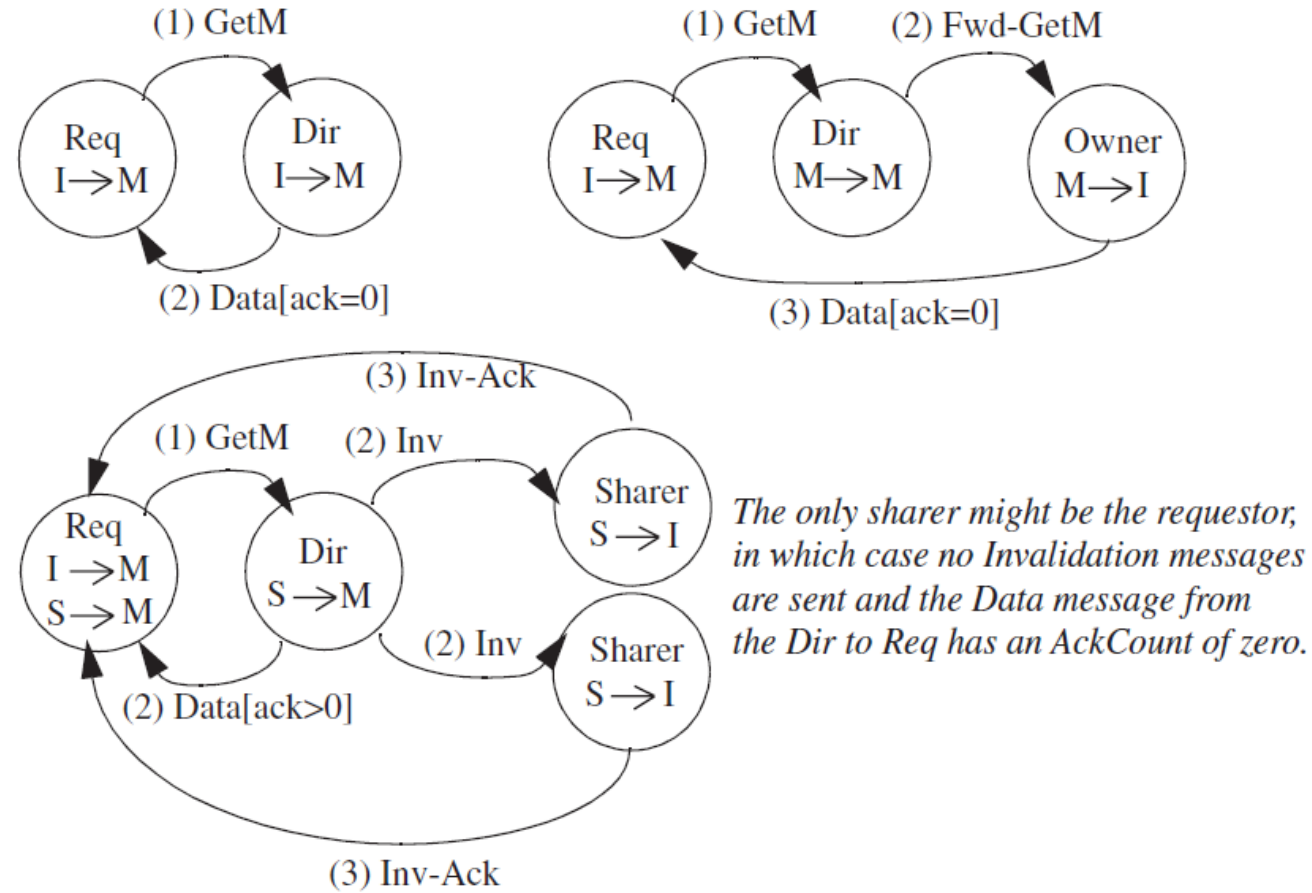
# MSI Directory Protocol



**Transitions from I to S.**

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

# MSI Directory Protocol



*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**

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

# MSI Directory Protocol

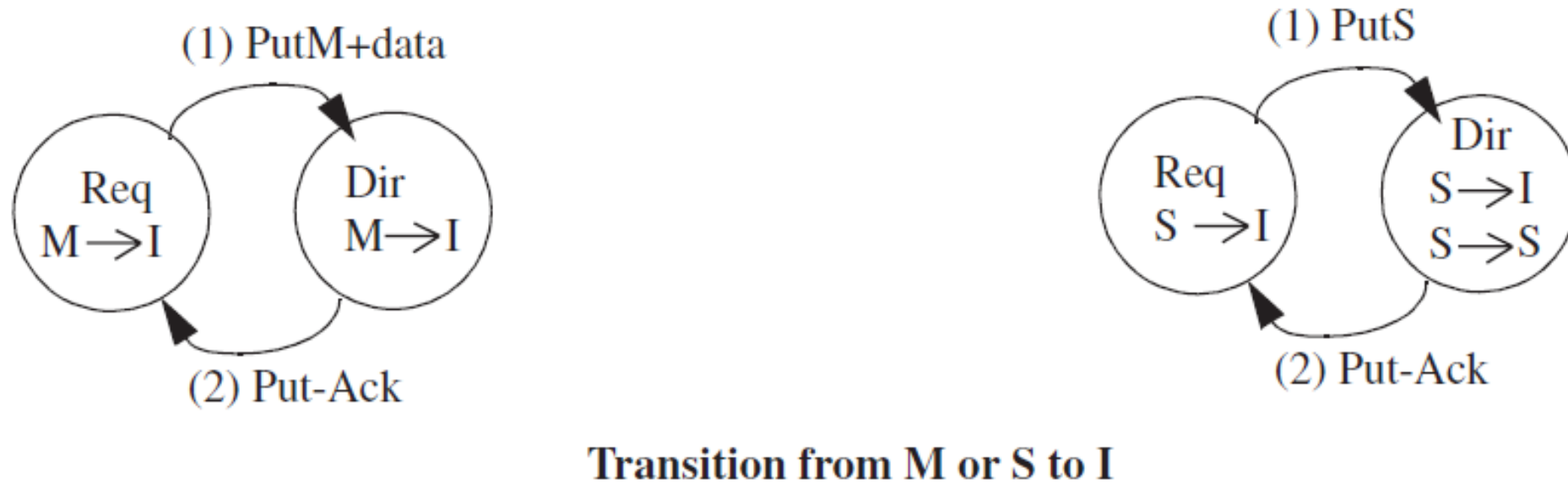


Fig 8.3 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.



# MESI Directory Protocol

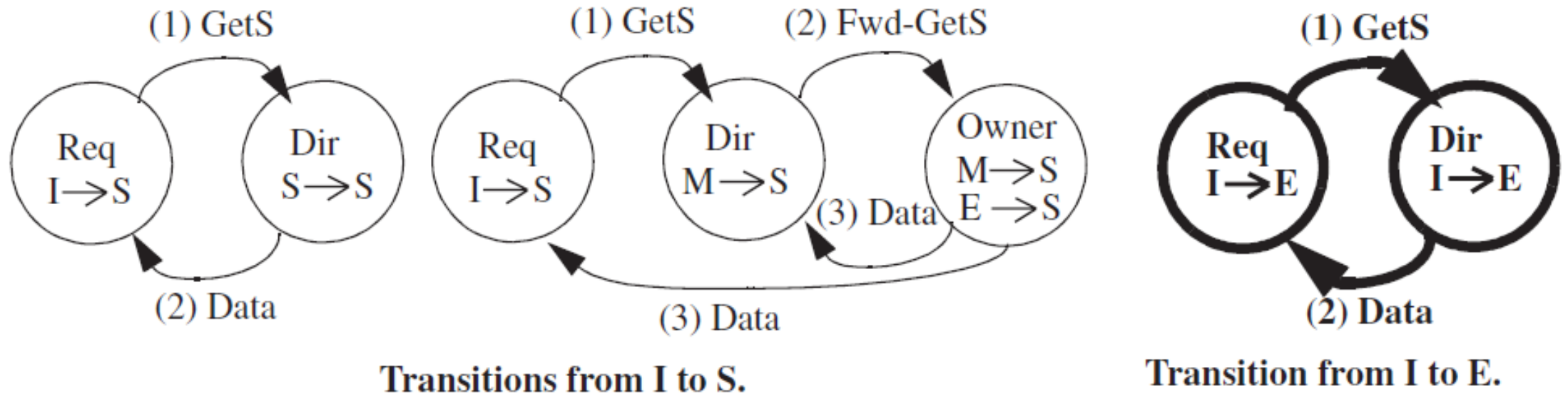
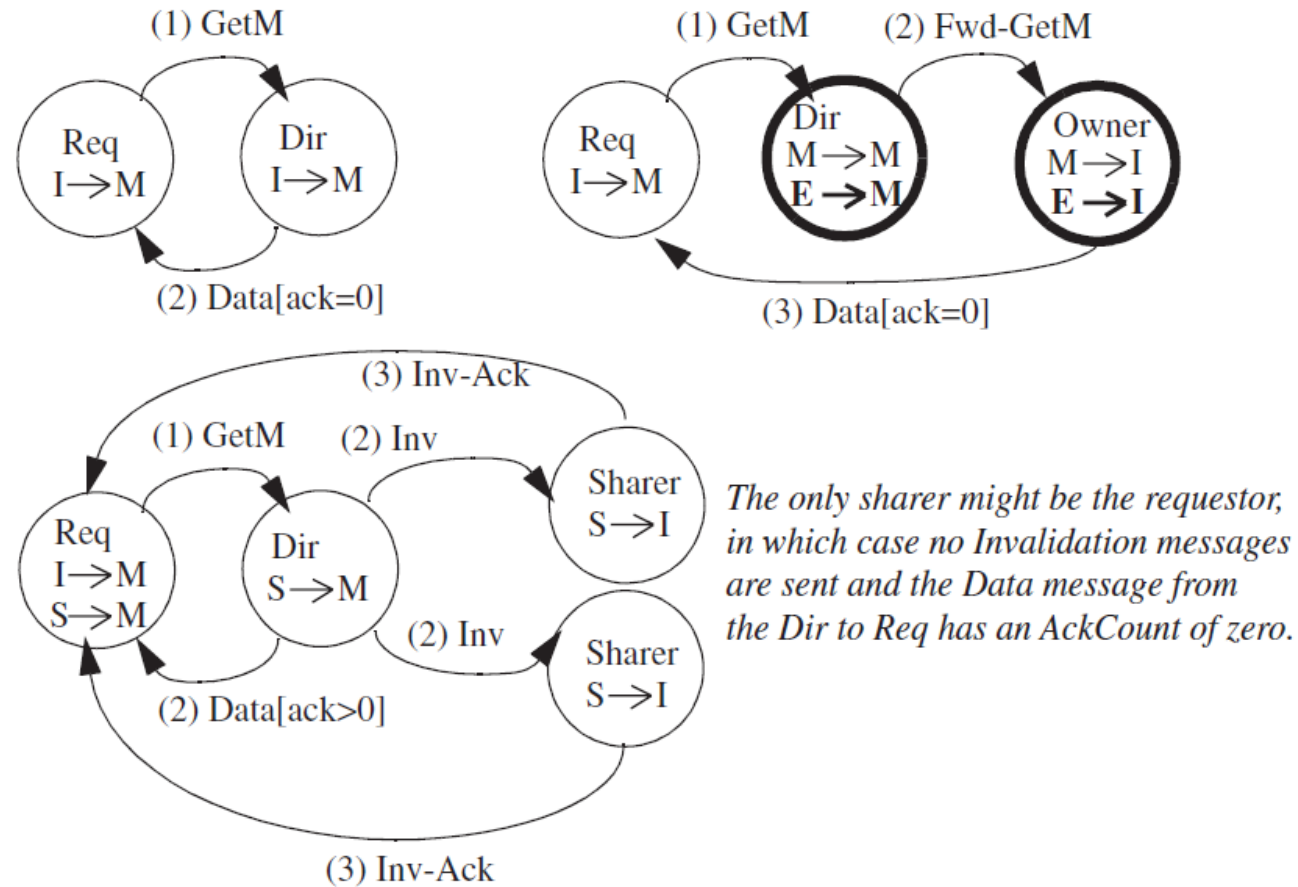


Fig 8.6 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

# MESI Directory Protocol



**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

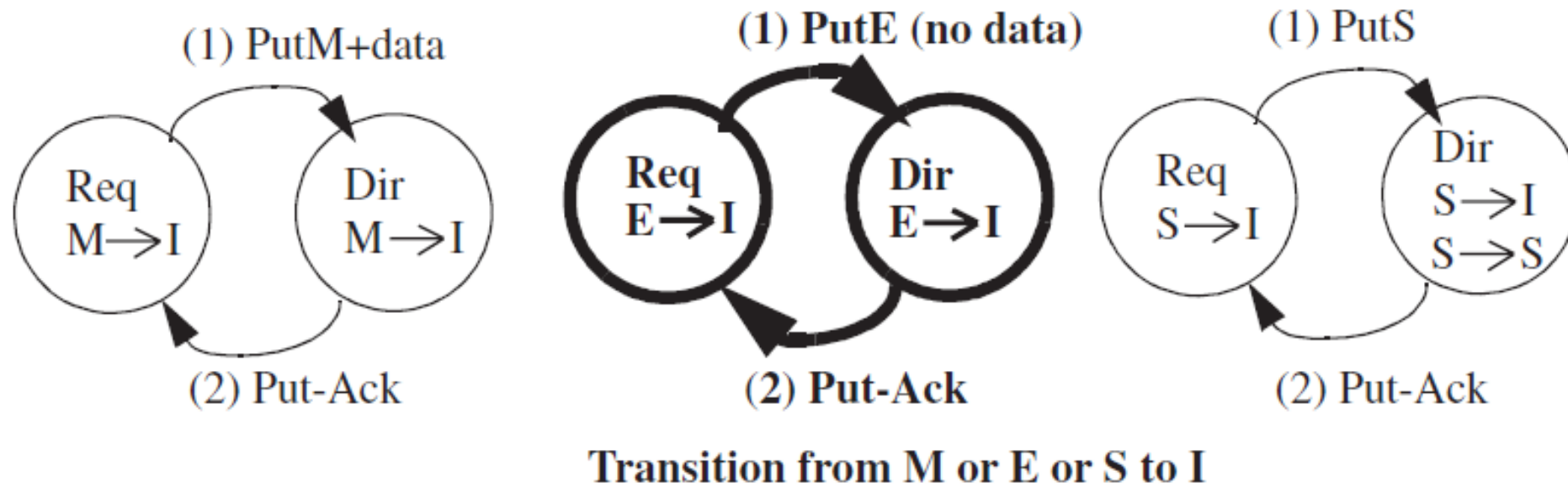


Fig 8.6 from D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.

# Cache Contention

## True Sharing

- Same location is accessed by multiple cores
- Fixed by means of algorithmic changes

## False Sharing

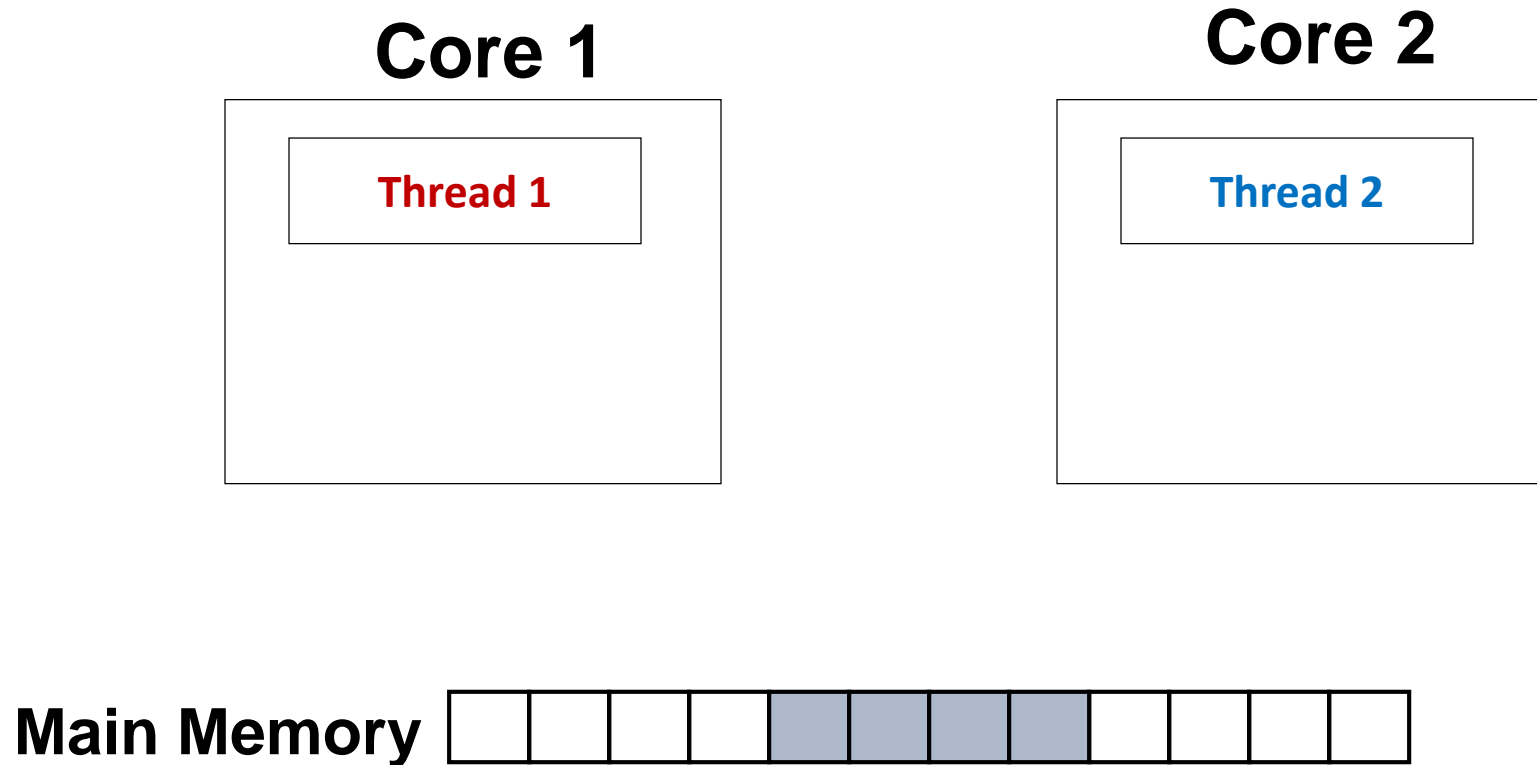
- Two unrelated locations share a cache line
- Fixed by code changes or by automatic repair

# What is False Sharing?

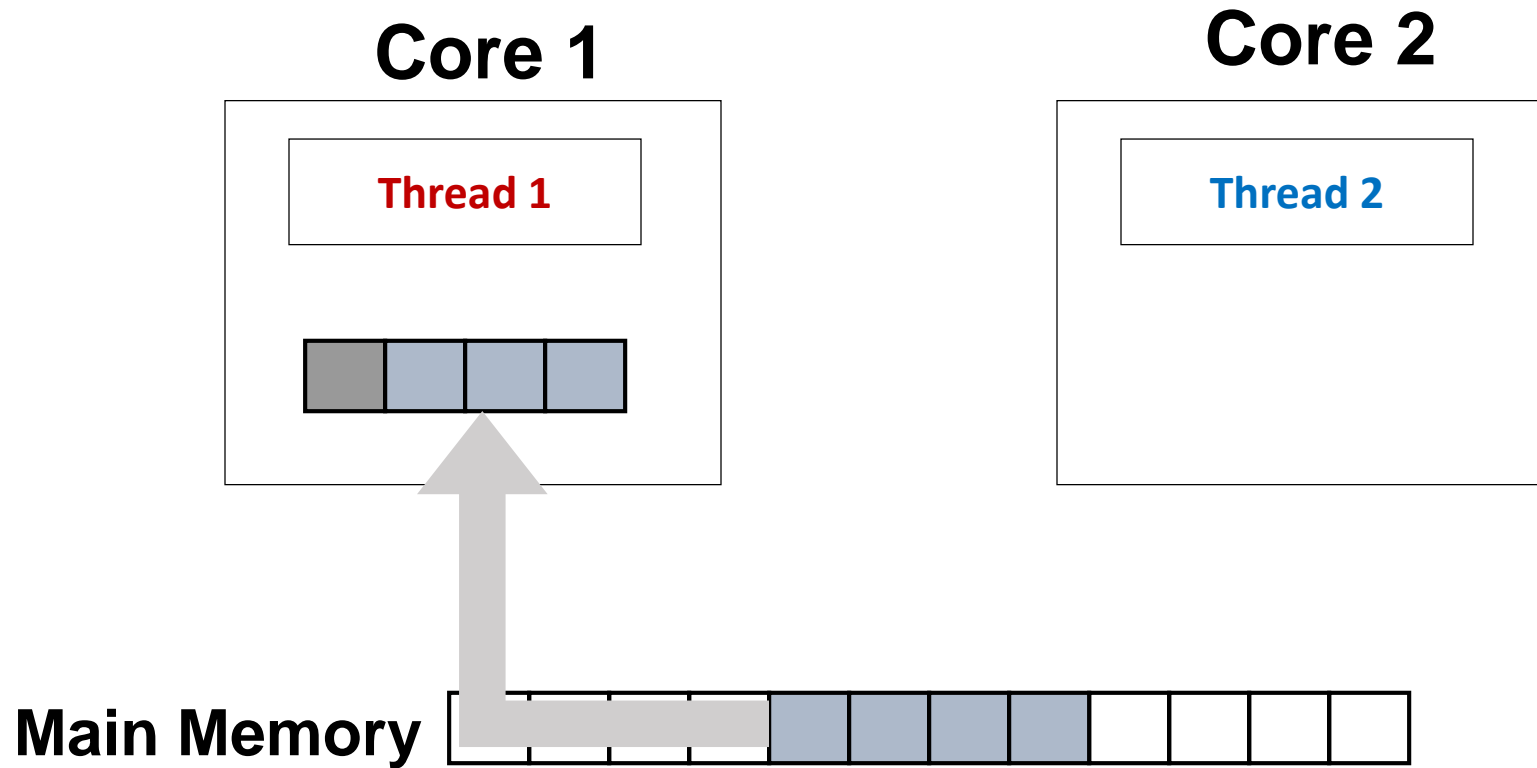
- Performance problem in systems with coherence caches
  - Cores share cache blocks instead of actual data
  - Contention on cache lines
- Can arise when threads access global or heap memory
  - Thread-local storage and local variables can be ignored
- False sharing is aggravated by the size of cache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



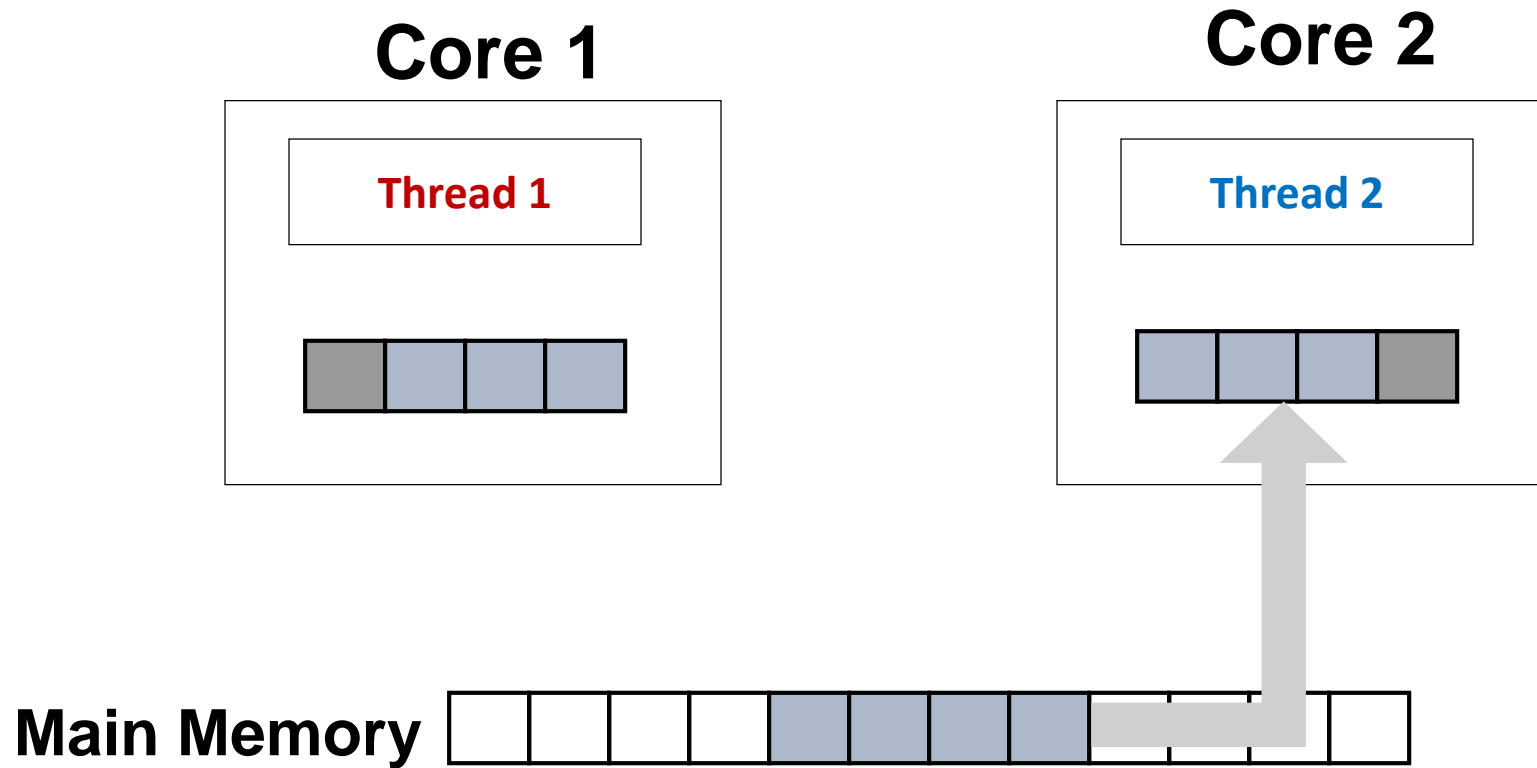
# False Sharing: A Performance Problem



# False Sharing: A Performance Problem

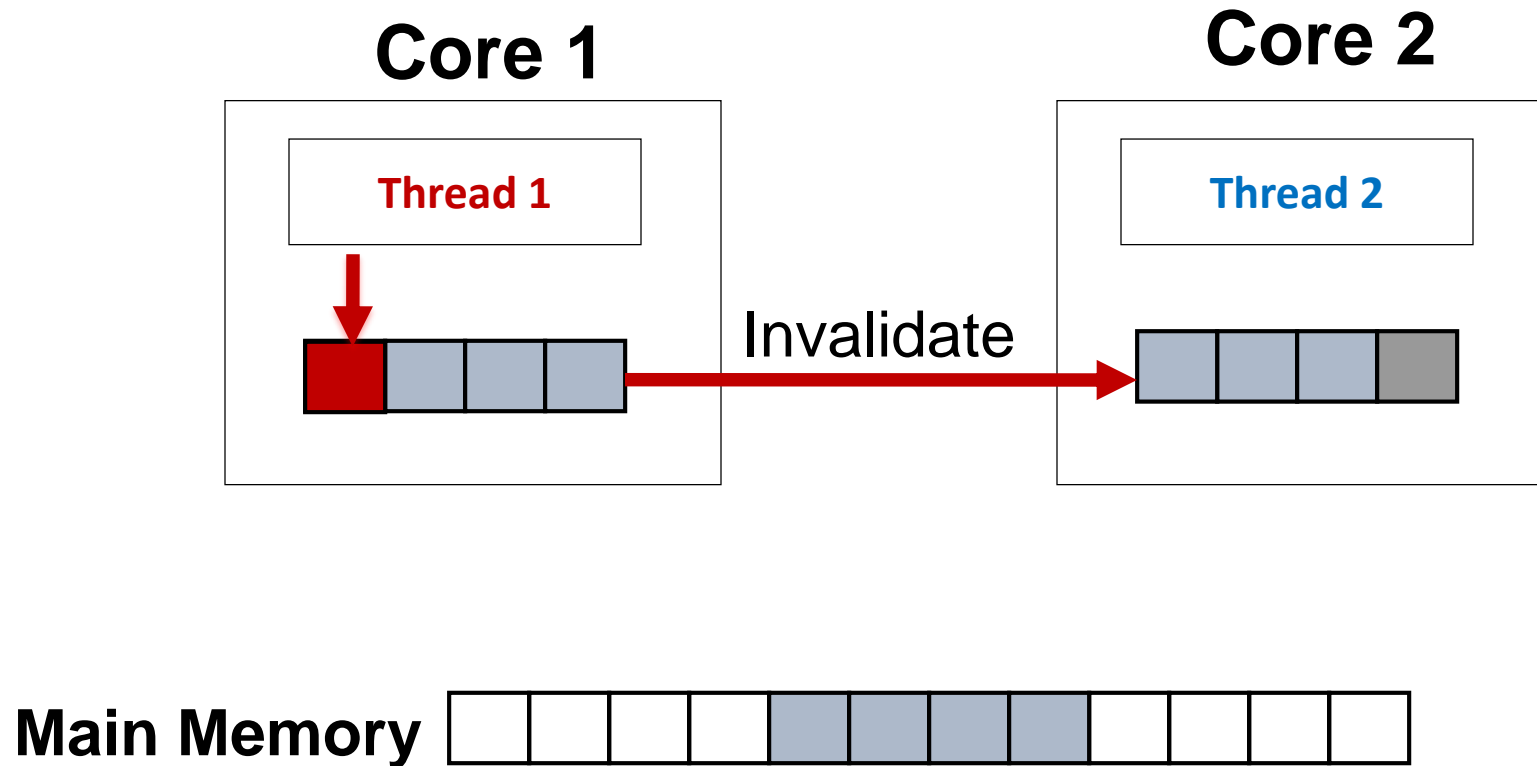


# False Sharing: A Performance Problem

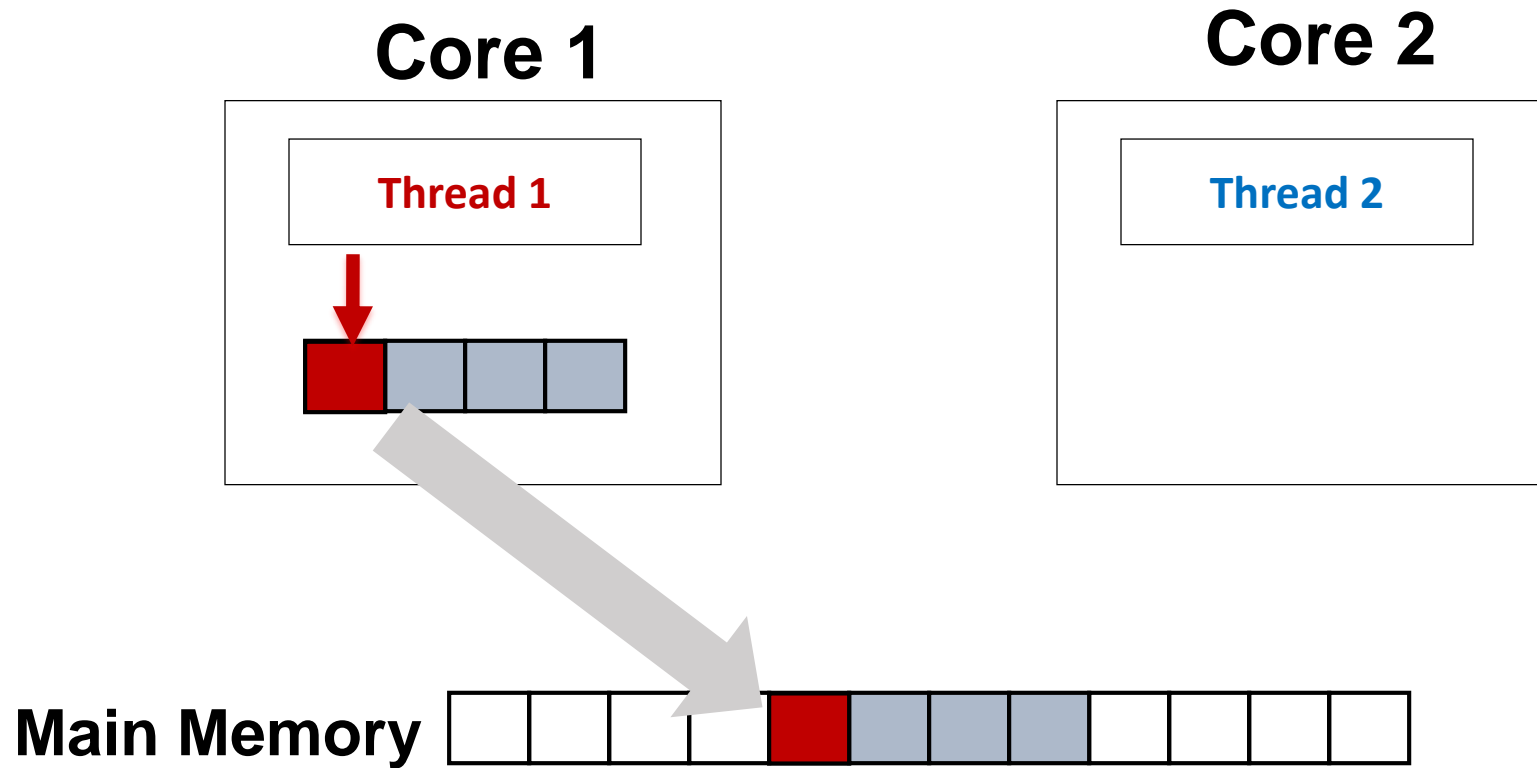




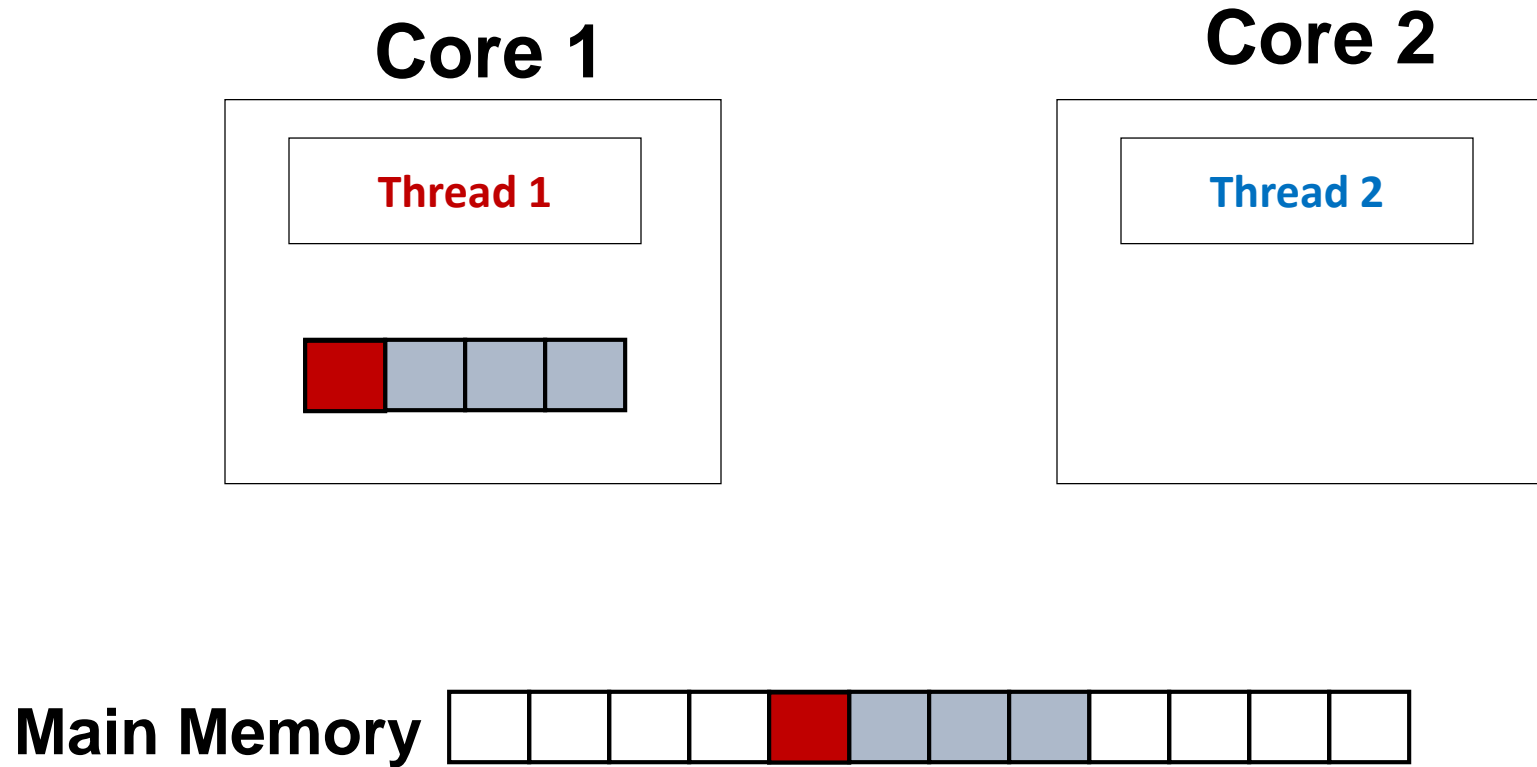
# False Sharing: A Performance Problem



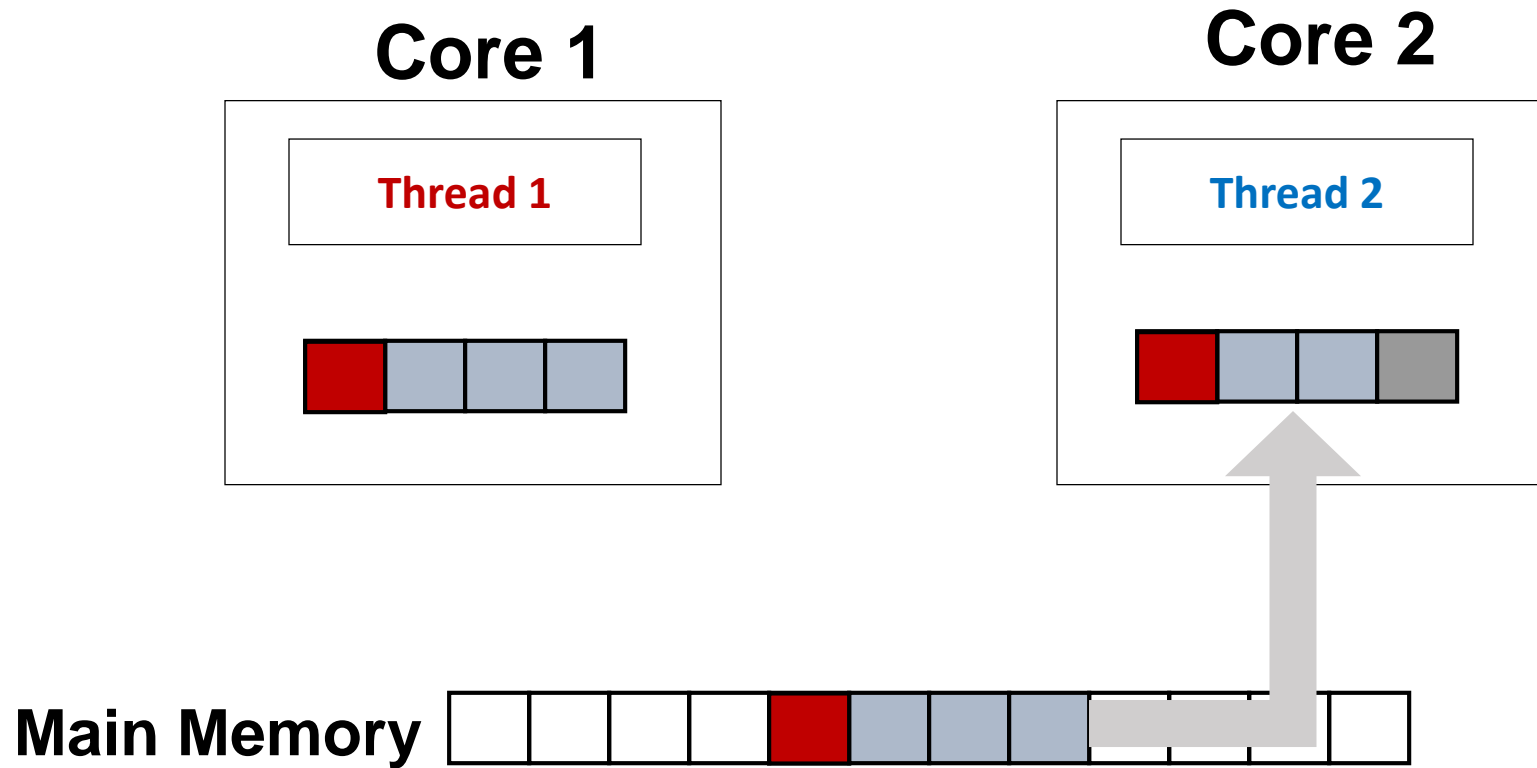
# False Sharing: A Performance Problem



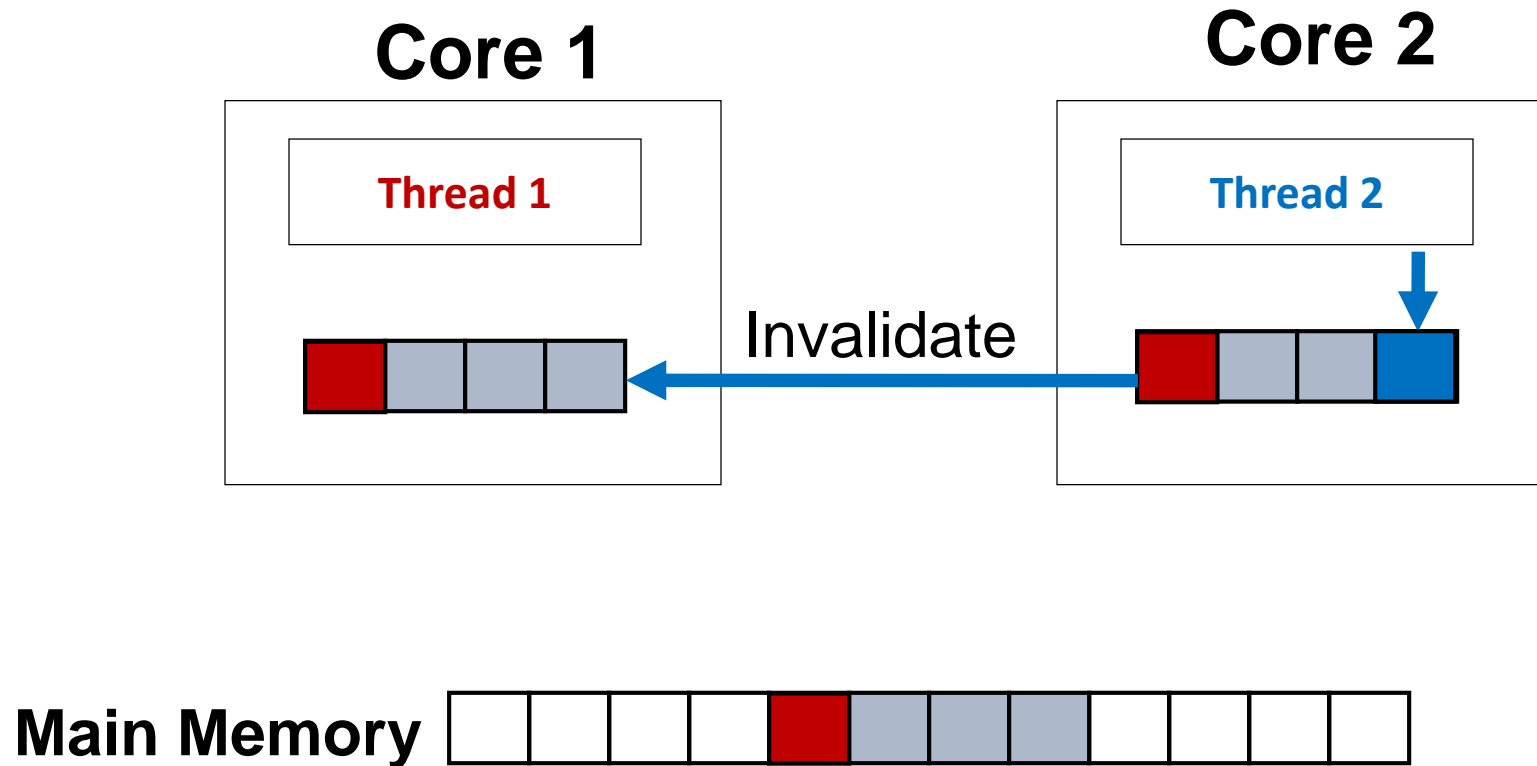
# False Sharing: A Performance Problem



# False Sharing: A Performance Problem



# False Sharing: A Performance Problem



# 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, ...);
}
```

# 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
    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, ...);

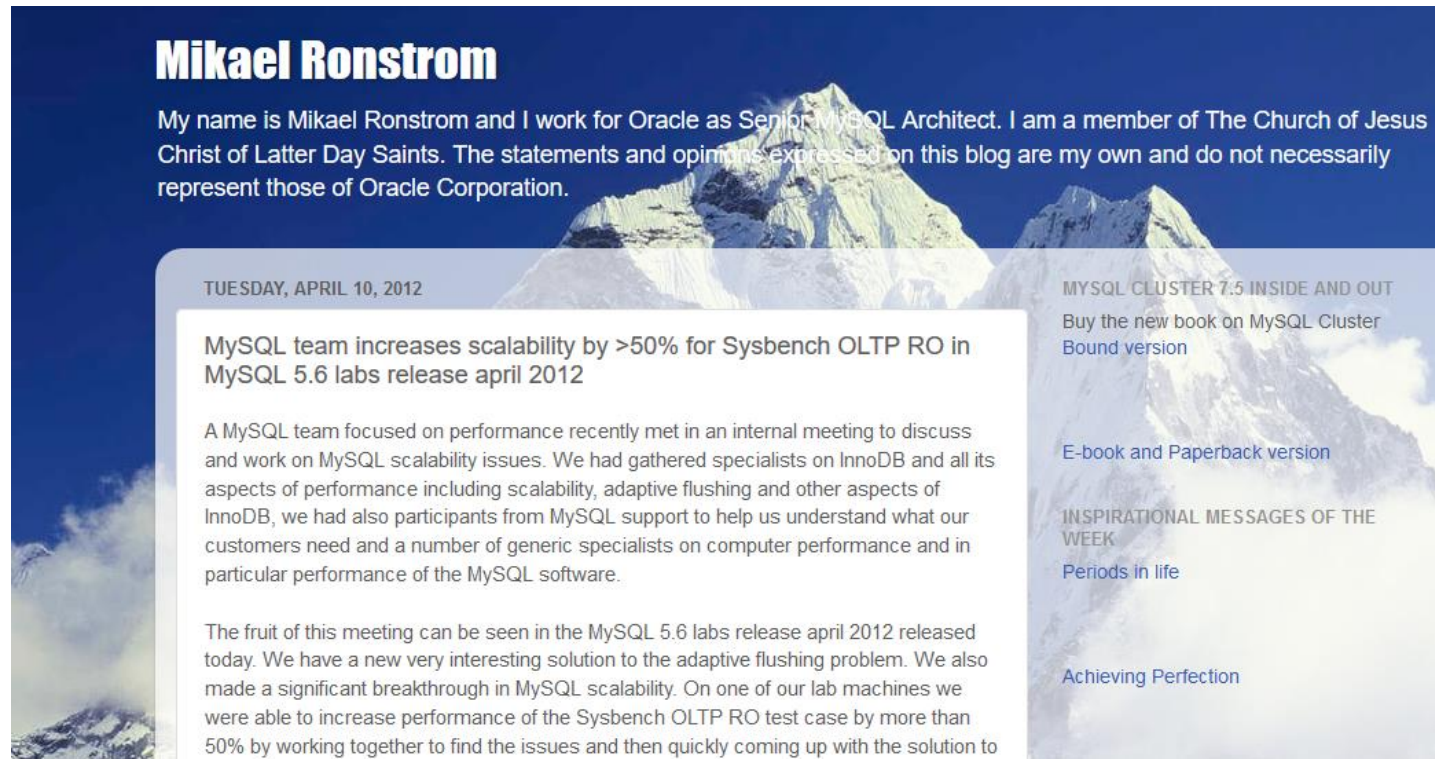
clock_gettime(CLOCK_REALTIME, ...);
pthread_create(&thread_1, NULL, func, (void*)&first_elem);
pthread_create(&thread_2, NULL, func, (void*)&bad_elem);

pthread_join(thread_1, NULL);
pthread_join(thread_2, NULL);
clock_gettime(CLOCK_REALTIME, ...);
}
```

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

# False Sharing in Real Applications

- Issues reported in Linux kernel, JVM, Boost library, ...



**Mikael Ronstrom**

My name is Mikael Ronstrom and I work for Oracle as Senior MySQL Architect. I am a member of The Church of Jesus Christ of Latter Day Saints. The statements and opinions expressed on this blog are my own and do not necessarily represent those of Oracle Corporation.

TUESDAY, APRIL 10, 2012

### MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012

A MySQL team focused on performance recently met in an internal meeting to discuss and work on MySQL scalability issues. We had gathered specialists on InnoDB and all its aspects of performance including scalability, adaptive flushing and other aspects of InnoDB, we had also participants from MySQL support to help us understand what our customers need and a number of generic specialists on computer performance and in particular performance of the MySQL software.

The fruit of this meeting can be seen in the MySQL 5.6 labs release april 2012 released today. We have a new very interesting solution to the adaptive flushing problem. We also made a significant breakthrough in MySQL scalability. On one of our lab machines we were able to increase performance of the Sysbench OLTP RO test case by more than 50% by working together to find the issues and then quickly coming up with the solution to

MYSQL CLUSTER 7.5 INSIDE AND OUT  
Buy the new book on MySQL Cluster  
[Bound version](#)

[E-book and Paperback version](#)

INSPIRATIONAL MESSAGES OF THE WEEK  
[Periods in life](#)

[Achieving Perfection](#)


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



# False Sharing is Everywhere

## // Global variables

```
me = 1;  
you = 2;
```



different  
threads

## // Class/struct fields

```
class X {  
    int me;  
    float you;  
};
```

## // Heap objects

```
me = new Foo();  
you = new Bar();
```

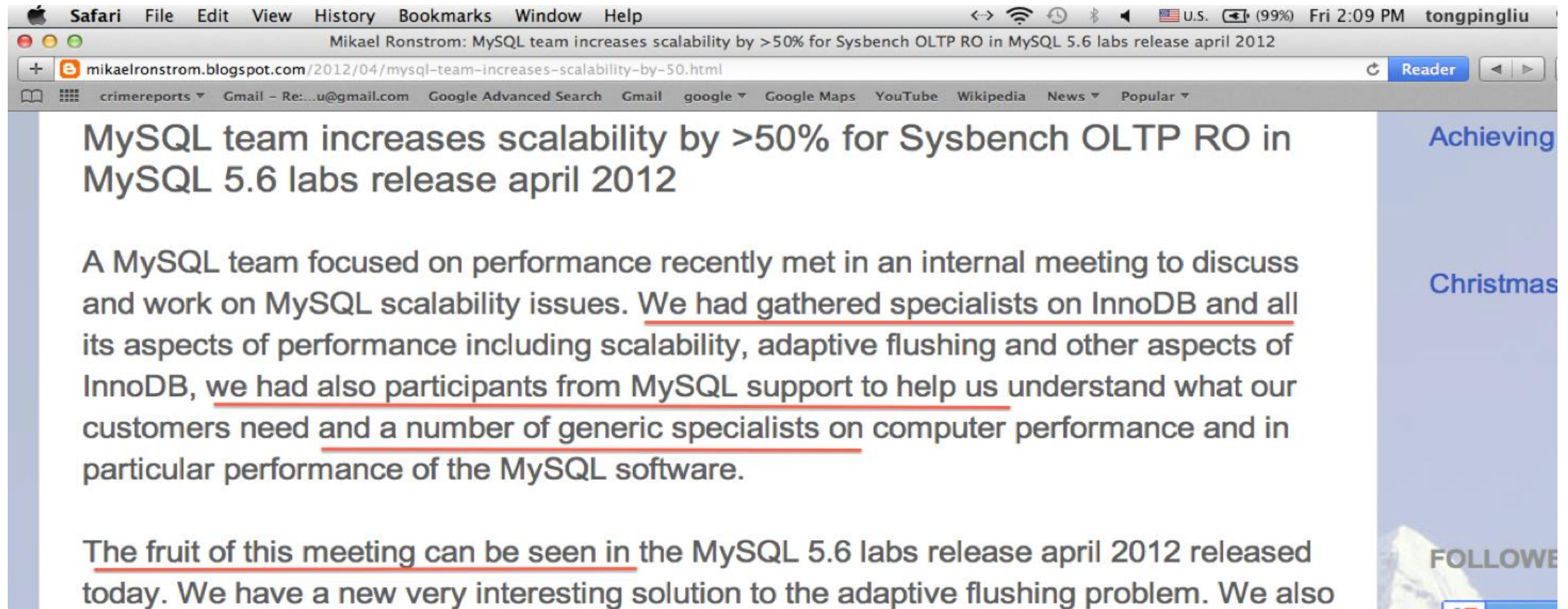
## // Array accesses

```
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



The screenshot shows a Safari browser window on a Mac. The address bar displays the URL: [mikaelsonstrom.blogspot.com/2012/04/mysql-team-increases-scalability-by-50.html](http://mikaelsonstrom.blogspot.com/2012/04/mysql-team-increases-scalability-by-50.html). The page title is "MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012". The main content of the blog post is as follows:

**MySQL team increases scalability by >50% for Sysbench OLTP RO in MySQL 5.6 labs release april 2012**

A MySQL team focused on performance recently met in an internal meeting to discuss and work on MySQL scalability issues. We had gathered specialists on InnoDB and all its aspects of performance including scalability, adaptive flushing and other aspects of InnoDB, we had also participants from MySQL support to help us understand what our customers need and a number of generic specialists on computer performance and in particular performance of the MySQL software.

The fruit of this meeting can be seen in the MySQL 5.6 labs release april 2012 released today. We have a new very interesting solution to the adaptive flushing problem. We also

On the right side of the browser window, there is a sidebar with a blue background. It contains the text "Achieving" and "Christmas" in blue font. At the bottom of the sidebar, there is a "FOLLOW" button with a small icon of a person.

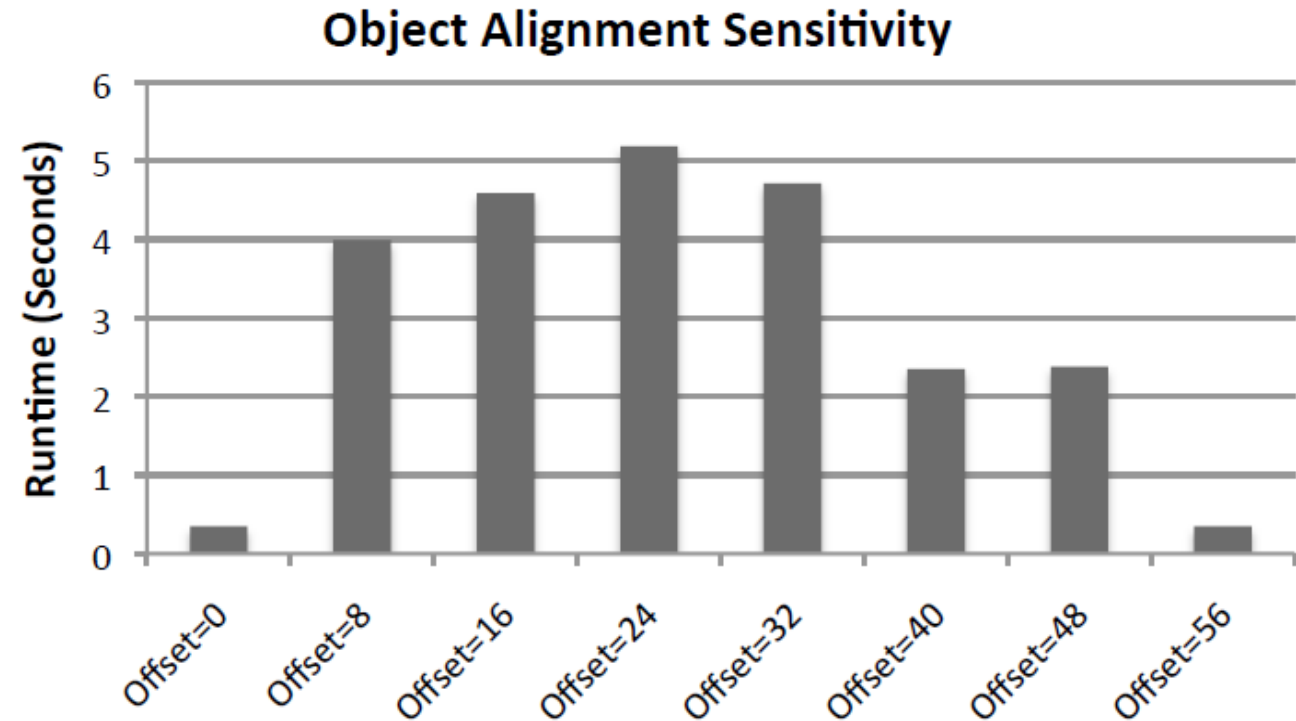
# Fixing False Sharing is Non-trivial

- 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 unintentionally eliminates false sharing in Phoenix  
linear\_regression benchmark at certain optimization levels, while  
LLVM does not do so at any optimization level

# Object Alignment Sensitivity

- Plot shows the performance of the linear\_regression benchmark from the Phoenix benchmark suite
- Performance is highly sensitive to the offset of the starting address of the (potentially) falsely-shared object from the start of the cache line



# Research on False Sharing Detection and Repair

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	
DeFT	Venkataramani et al., TACO'11	
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	
Huron	Khan et al., PLDI'19	Prioritizes static time repair

# 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-contented-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>  
<http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.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>  
<http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.html>

# Sheriff: Precise Detection and Automatic Mitigation

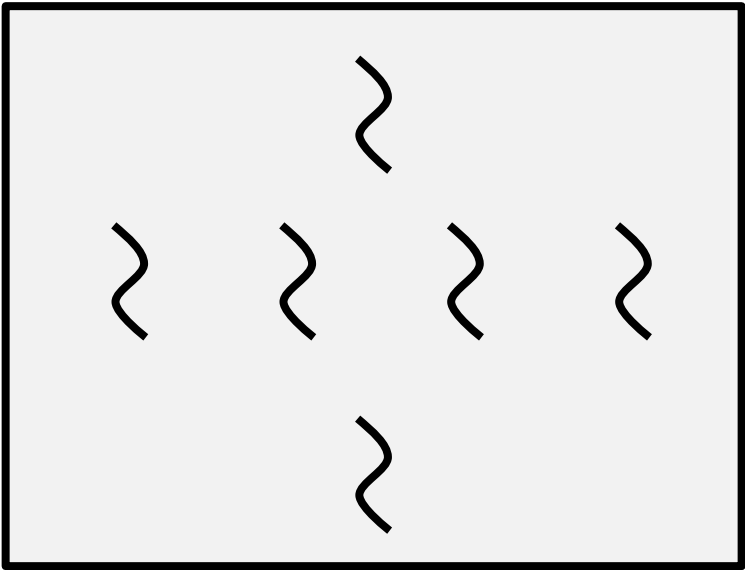
- Sheriff is a software-only solution that provides
  - Per-thread memory protection – allows each thread to track memory accesses independently of other thread's accesses
  - Memory isolation – allows each thread to read from and write to memory without interference from other threads

---

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

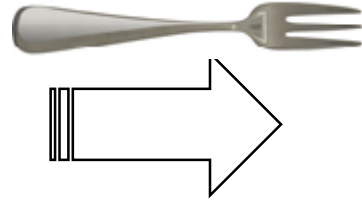
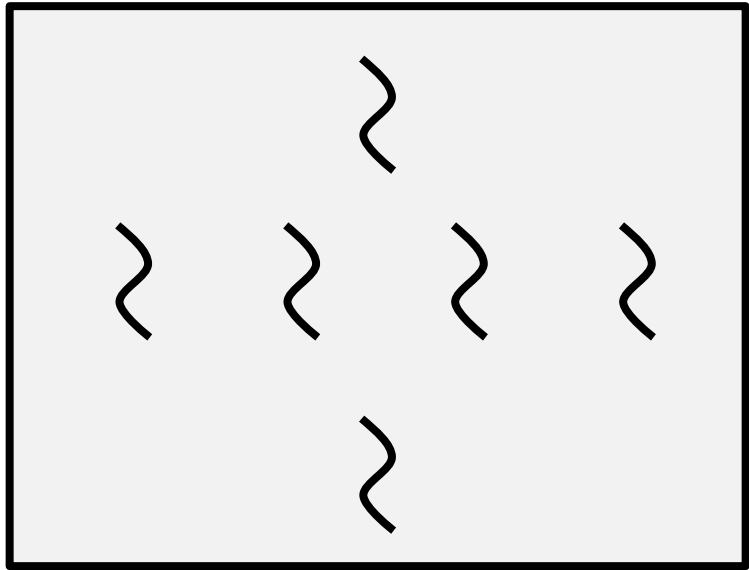
# Isolated Memory Access

shared address space

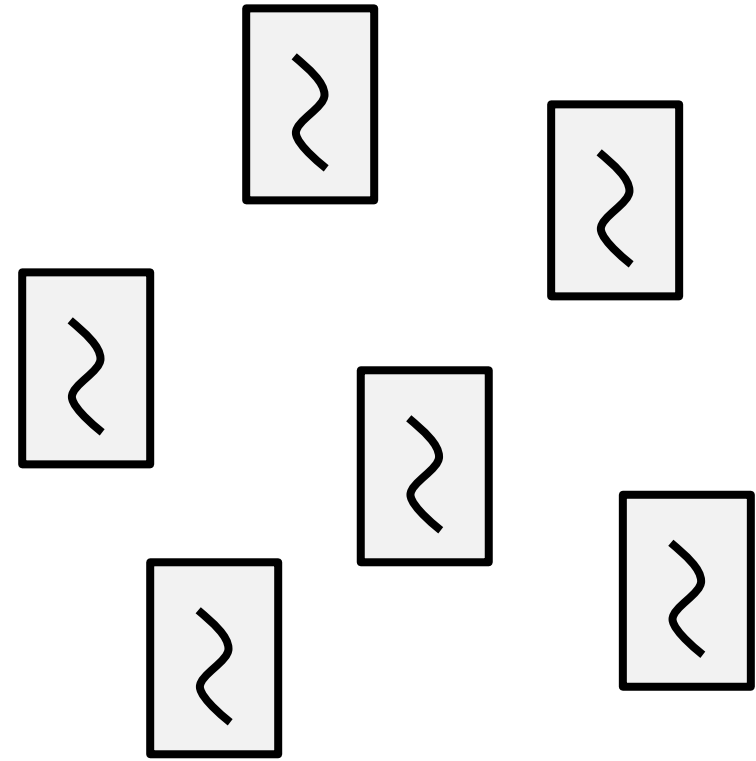


# Isolated Memory Access

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 – Delaying updates can avoid false sharing, accesses will no longer be concurrent
- Protects small objects
  - Benefit of protection is greater than large objects like arrays (relative to the object size)
  - 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-based compiler instrumentation 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

**Compiler  
Instrumentation**

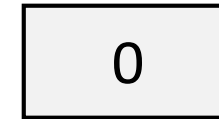
**Runtime System**

---

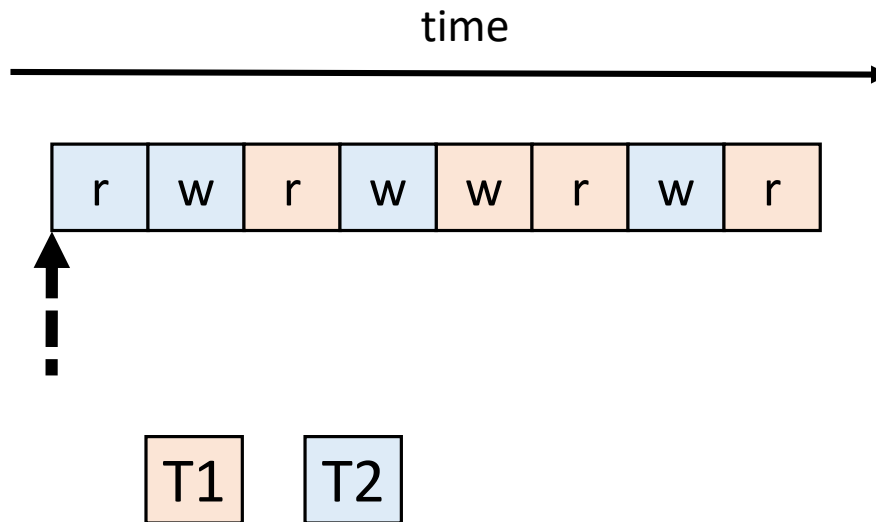
T. Liu et al. Predator: Predictive False Sharing Detection. PPOPP 2014.

# Track Cache Invalidations

Each Entry: {Thread ID, Access Type}



**# of invalidations**



# Rules for Per-Cache-Line History Table

- For each read  $R$ ,
  - If history 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  $W$ .

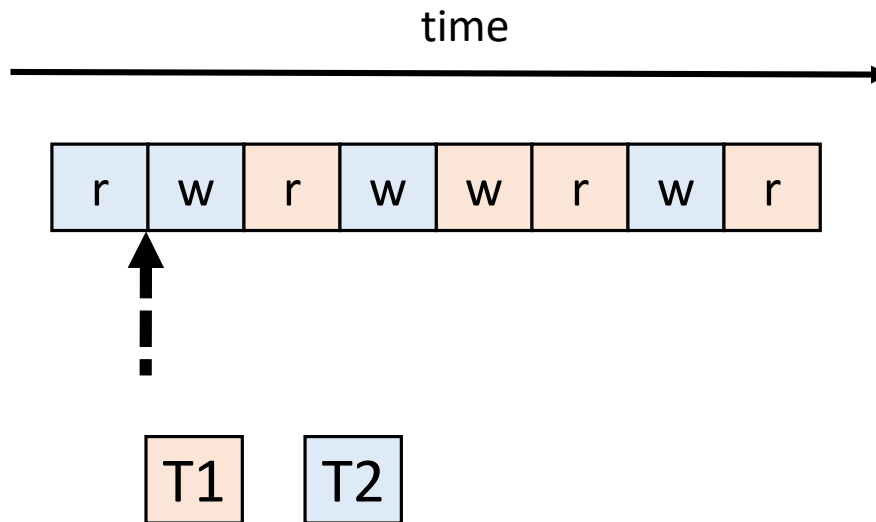
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T2	r	0	0
----	---	---	---

0
---

**# of invalidations**





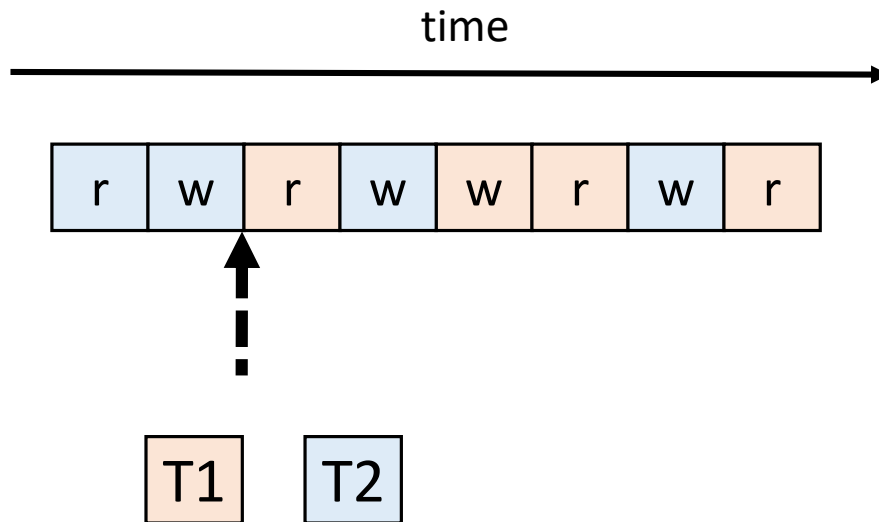
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T2	w	0	0
----	---	---	---

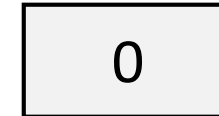
0
---

**# of invalidations**

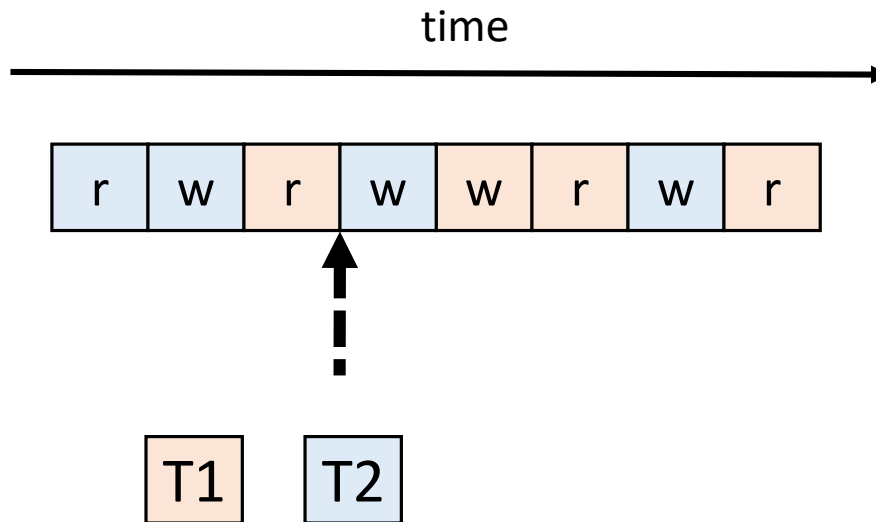


# Track Cache Invalidations

Each Entry: { Thread ID, Access Type}



**# of invalidations**



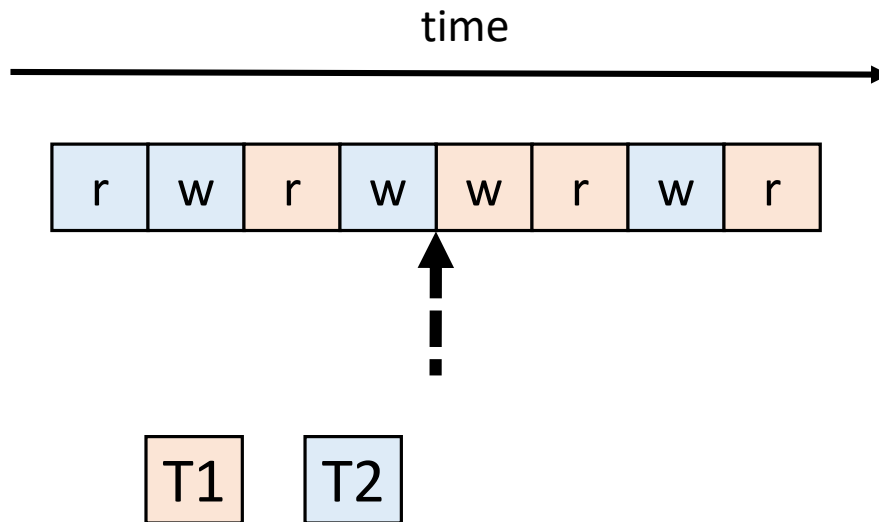
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T2	w	0	0
----	---	---	---

1
---

**# of invalidations**



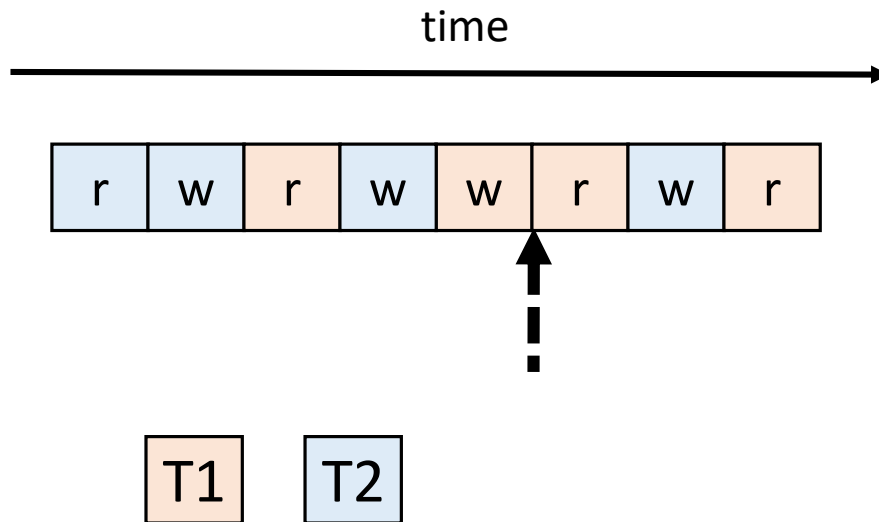
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T1	w	0	0
----	---	---	---

2
---

**# of invalidations**



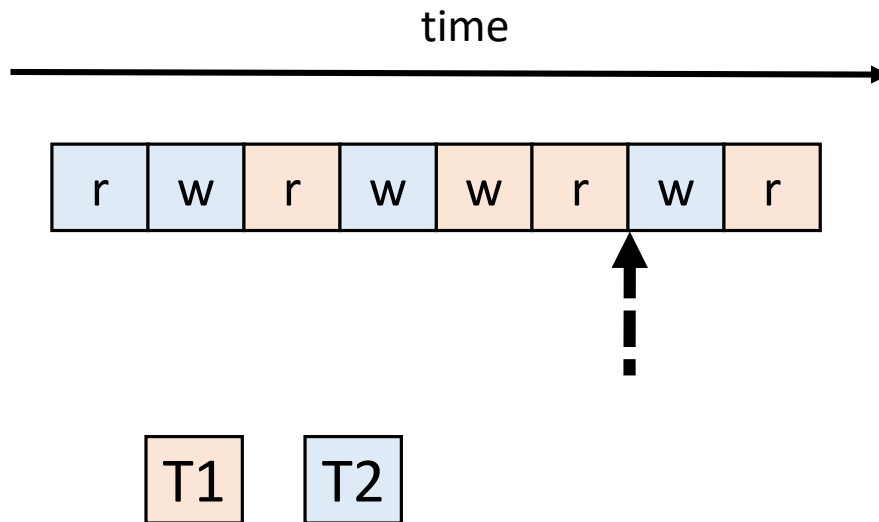
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T1	w	0	0
----	---	---	---

2
---

**# of invalidations**



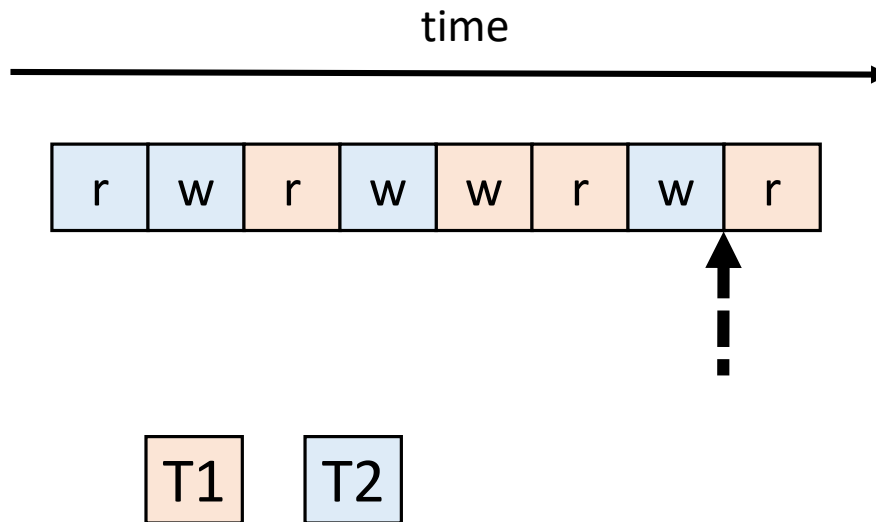
# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }

T2	w	0	0
----	---	---	---

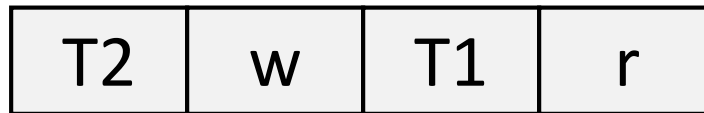
3
---

**# of invalidations**

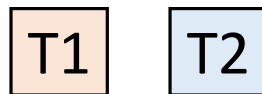
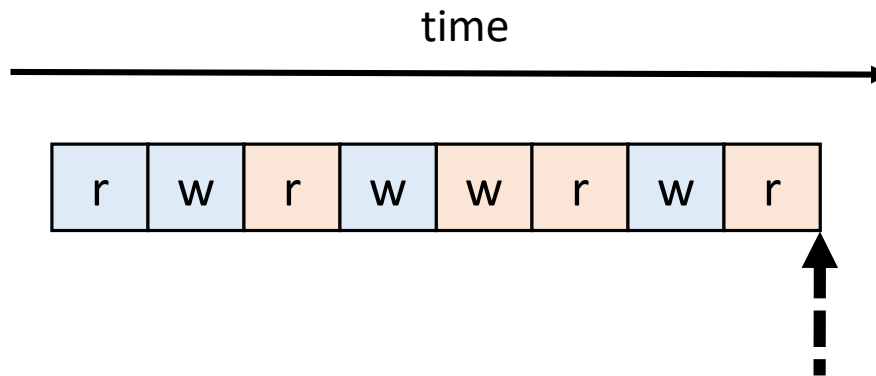


# Track Cache Invalidations

Each Entry: { Thread ID, Access Type }



**# of invalidations**



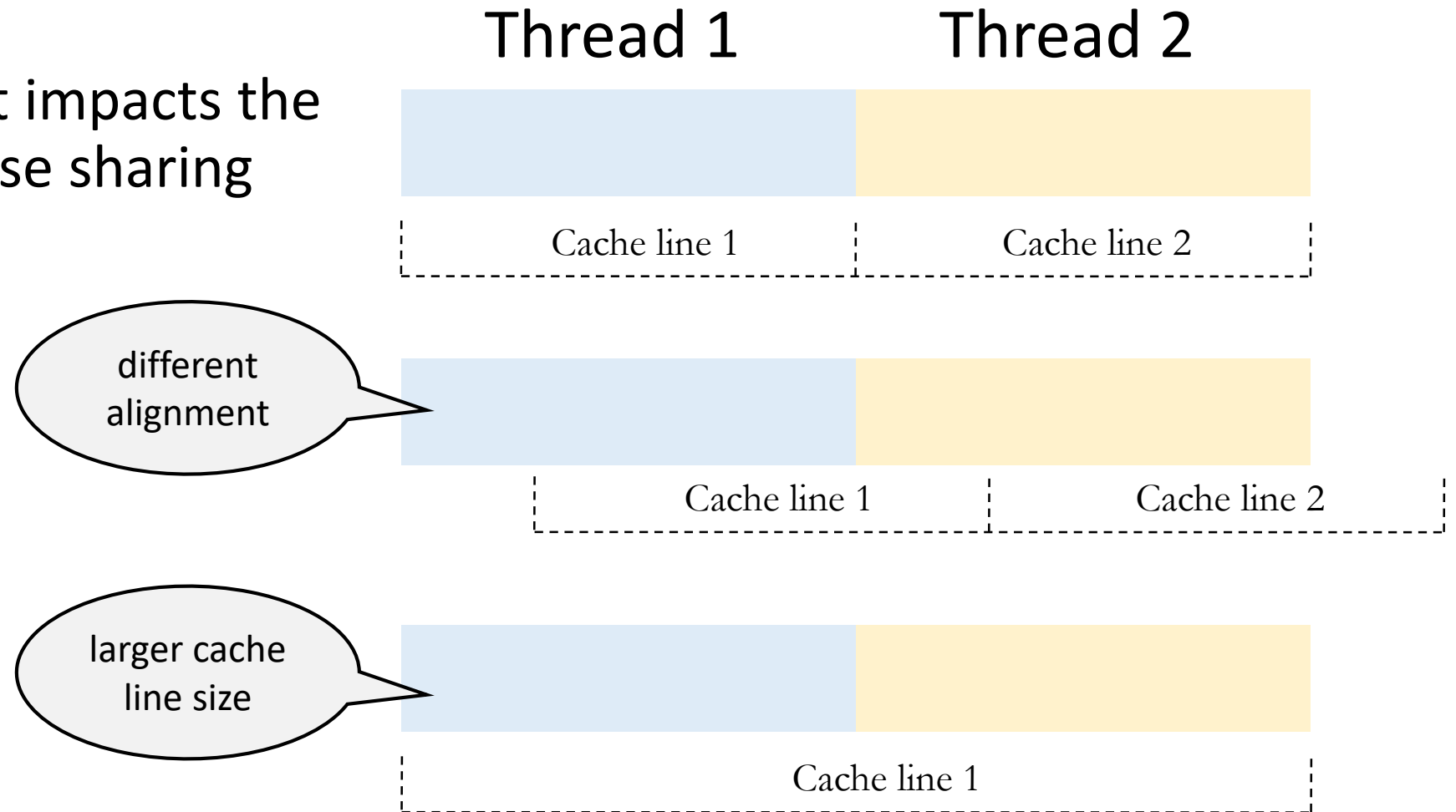
# Is that it?

- 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 sizes

# Prediction in Predator

- **Insight**

- Only accesses to adjacent lines can lead to potential false sharing

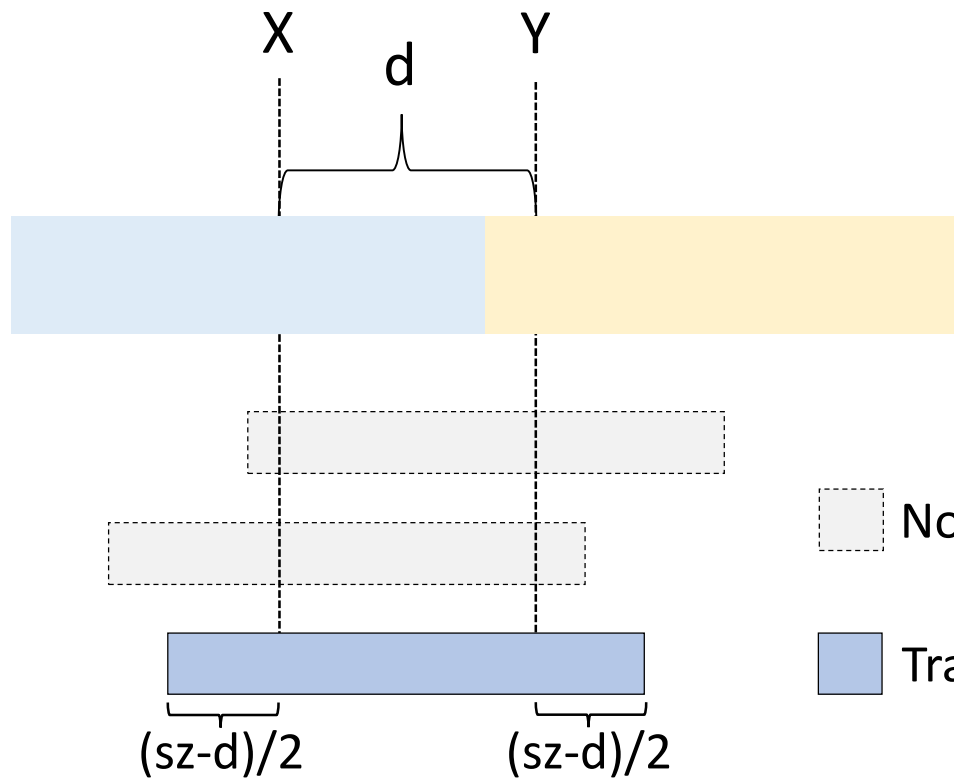
- Virtual cache line

- 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

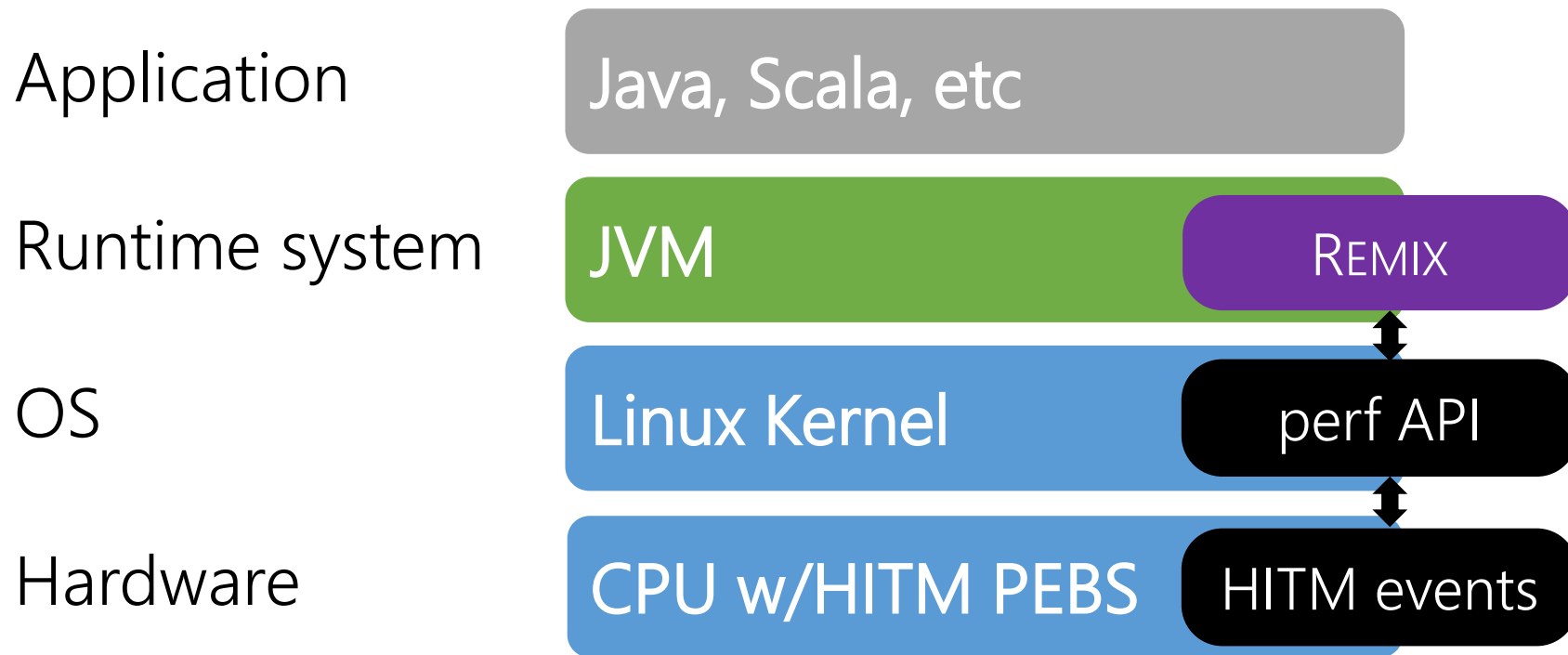


- $d < \text{cache line size } (sz)$
- X and Y are accesses from different threads
- One of X and Y accesses is a write

# False Sharing in the JVM

- **Less** programmer control over memory  $\Rightarrow$  more vulnerable to performance bugs
  - You might use padding, but unused fields may not be allocated at all
  - Furthermore, GC might move around objects
- Runtime has increased control over execution  $\Rightarrow$  opportunities for dynamic optimization
- `@Contended` helps avoid false sharing, but does not automatically detect sources of contention

# REMIX System Overview



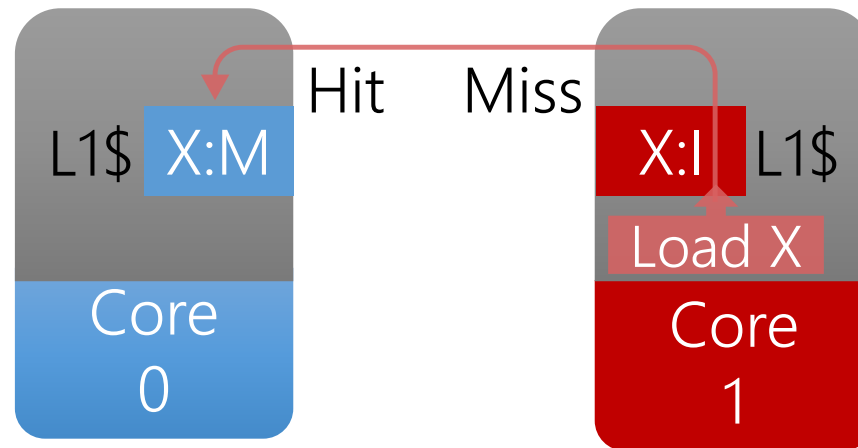
A. Eizenberg et al. Remix: Online Detection and Repair of Cache Contention for the JVM. PLDI 2016.

# Intel PEBS Events

- PEBS – Precise Event-Based Sampling
- Available in recent Intel multiprocessors
- Log detailed information about architectural events

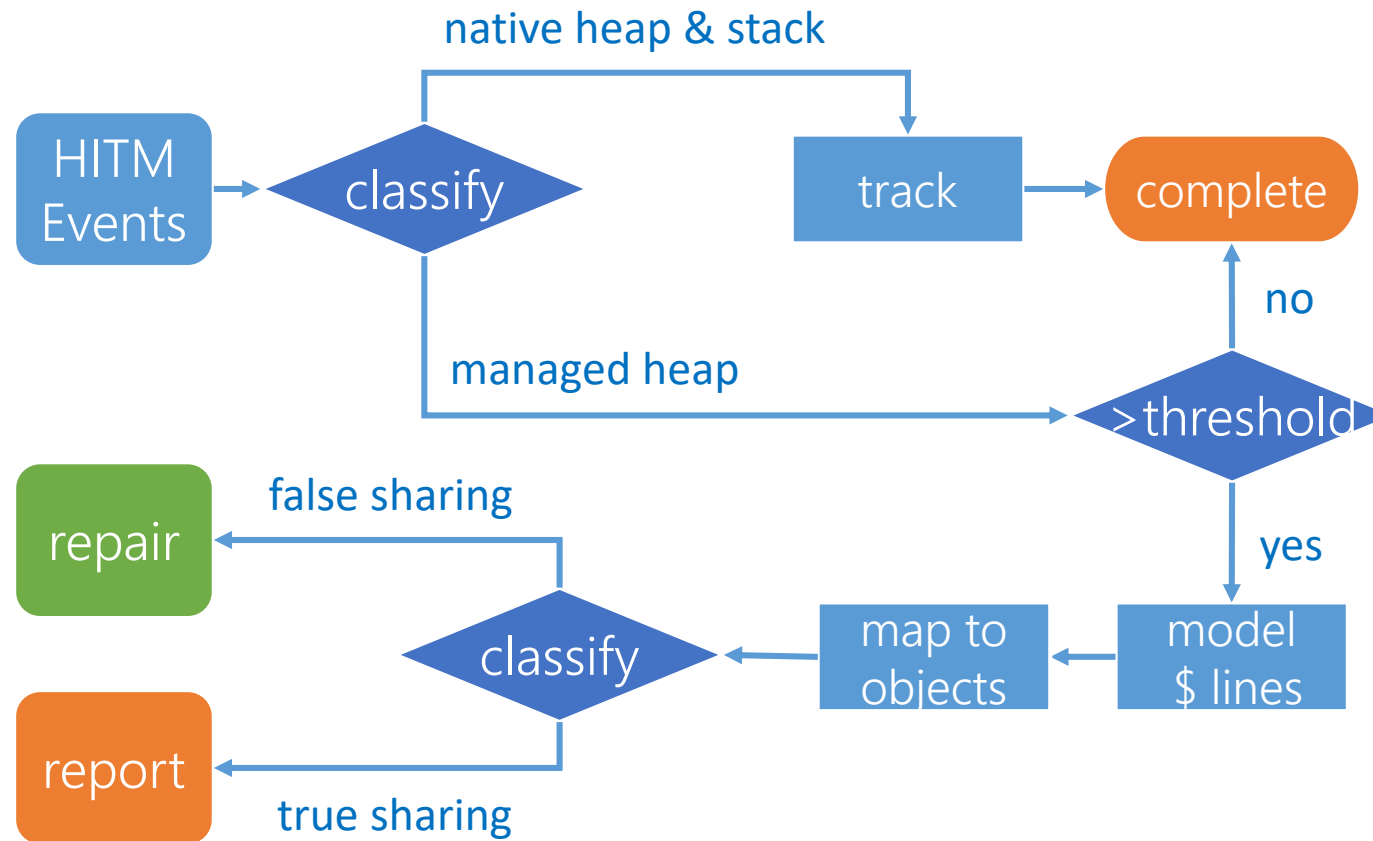
# PEBS HitM Events

- “**Hit-Modified**” - A cache miss due to a cache line in *Modified* state on a different core





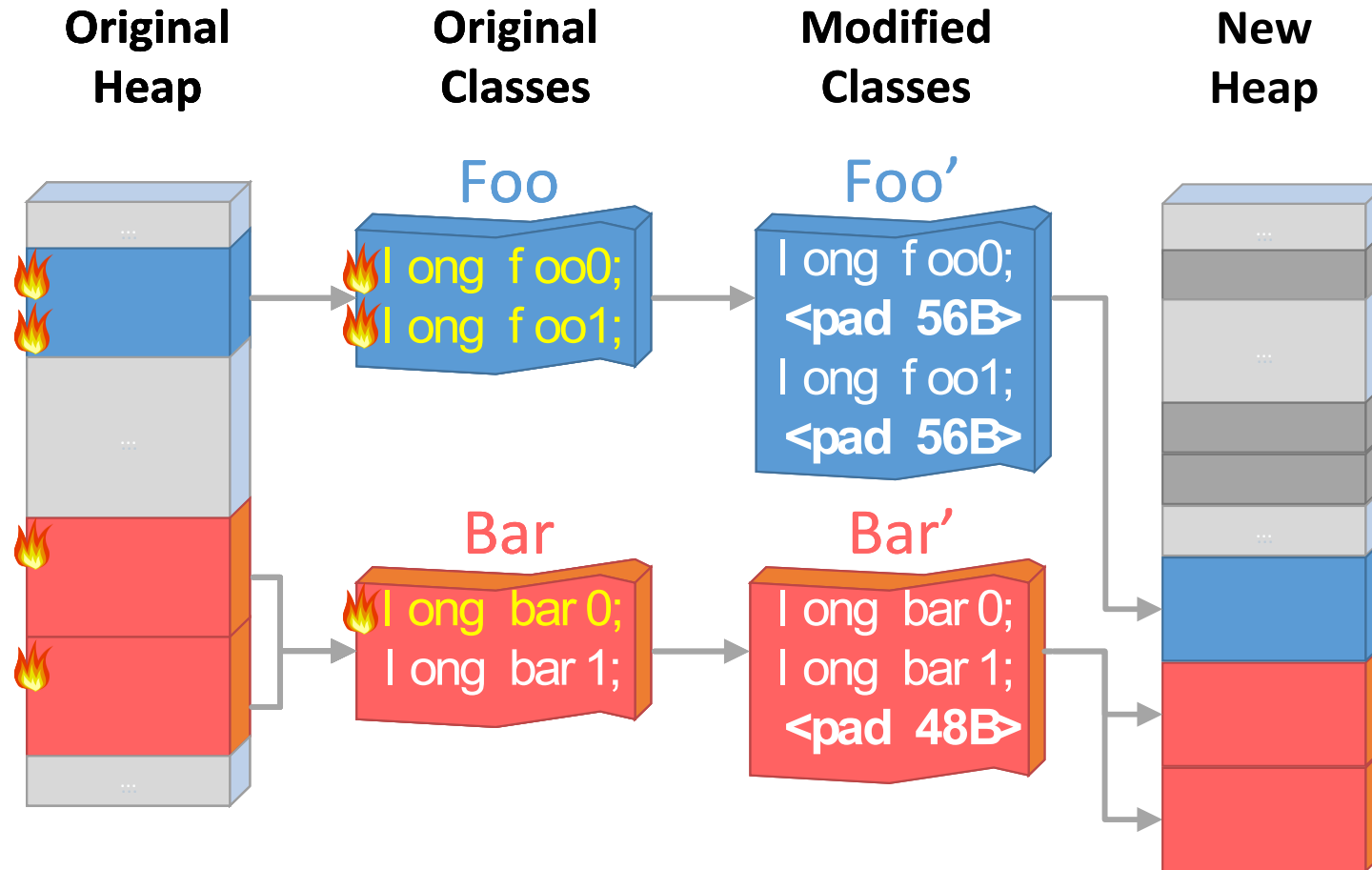
# Detection



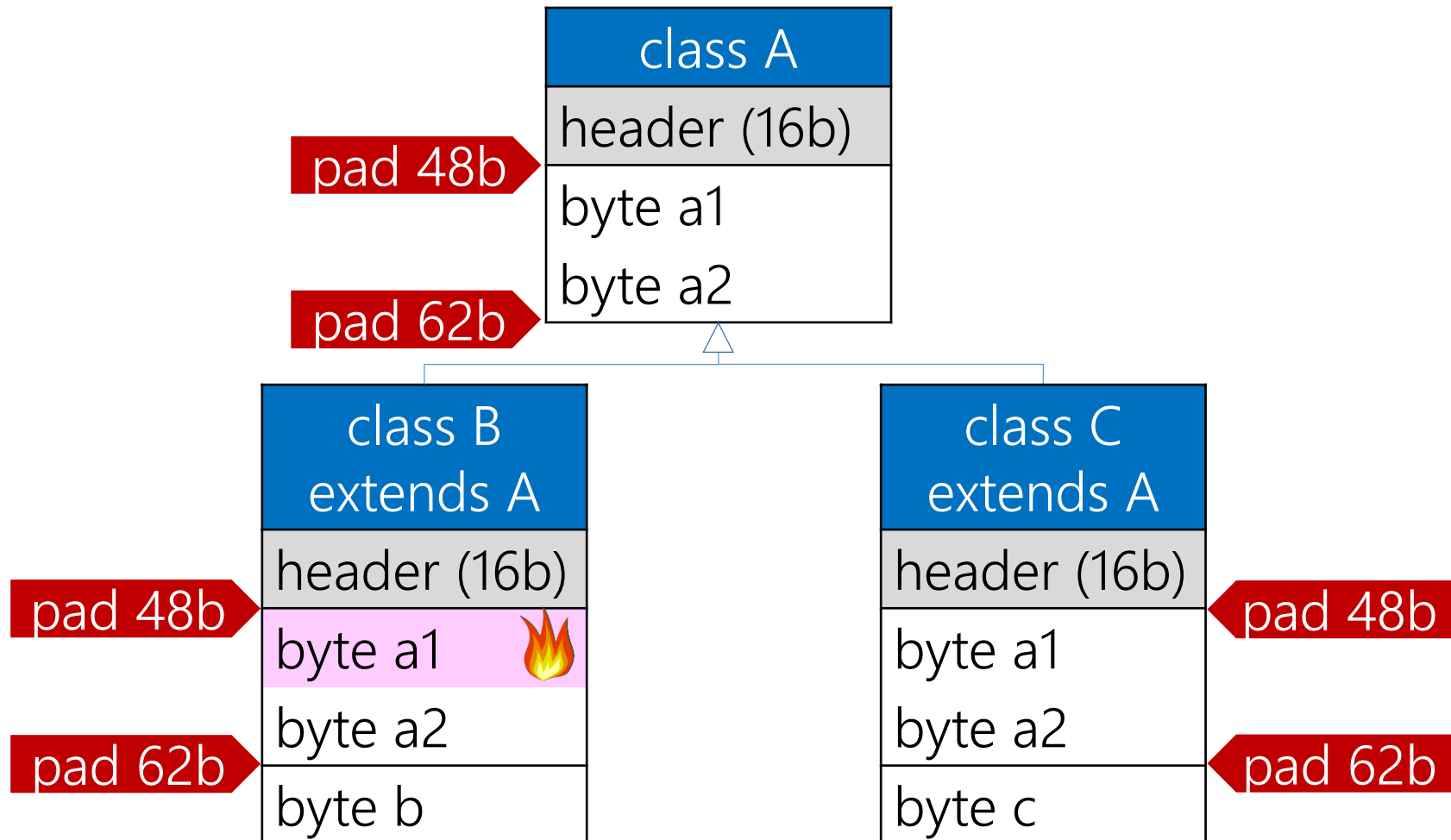
# Cache Line Modelling

- Cache lines are modelled with 64-bit bitmaps
- HITM event  $\Rightarrow$  set the address bit, count hit
- Multiple bits set  $\Rightarrow$  potential false sharing
- Repair is cheaper than more complete modelling!
- Repair when counter exceeds threshold

# Top Level Flow



# Padding - Inheritance



# Repair

- Trace all strong+weak roots in the system
- Traverse heap and find targeted instances
  - Live  $\Rightarrow$  Relocate & pad, store forwarding pointer
  - Dead  $\Rightarrow$  Fix size mismatch
- Adjust all pointers to forwarded objects
- Deoptimize all relevant stack frames

# References

- G. Jin et al. Understanding and Detecting Real-World Performance Bugs. PLDI 2012.
- T. Yu et al. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. ISSTA 2016.
- M. Alam et al. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. EuroSys 2017.
- M. Pradel et al. Performance Regression Testing of Concurrent Classes. ISSTA 2014.
- L. Zheng et al. On Performance Debugging of Unnecessary Lock Contentions on Multicore Processors: A Replay-based Approach. CGO 2015.
- R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. MICRO 2001.
- O. Mutlu. ECE 742: Parallel Computer Architecture, 2018. Lecture 16: Speculation.
- T. Liu and E. Berger. Sheriff: Precise Detection and Automatic Mitigation of False Sharing. OOPSLA 2011.
- T. Liu et al. Predator: Predictive False Sharing Detection. PPOPP 2014.
- A. Eizenberg et al. Remix: Online Detection and Repair of Cache Contention for the JVM. PLDI 2016.