CS 636: Performance of Concurrent Programs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-II

Evaluating Concurrent Programs

Functional correctness

- Does the application compute what it is supposed to do?
- Check for concurrency errors such as atomicity violations, order violations, sequential consistency violations, deadlocks, and livelocks

Performance correctness

- Does the application meet the performance requirements?
- Difficult to detect performance bottlenecks because of no failure symptoms
- Check for any performance regressions

Performance Testing

- 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), but software is getting more complex
  - Saving energy is now a primary concern

Performance Bug

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

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

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

A Thread-Unsafe Class in Groovy

```
1  class ExpandoMetaClass {
2    private boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.initialized = true;
6    }
7    boolean isInitialized() {
8      return this.initialized;
9    }
10 }
11
12
13
```

Before October 2007: Class ExpandoMetaClass is not thread-safe

```
1  class ExpandoMetaClass {
2    private boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.setInitialized(true);
6    }
7    synchronized void setInitialized(
         boolean b) {
8      this.initialized = b;
9    }
10   synchronized boolean isInitialized() {
11     return this.initialized;
12   }
13 }
```

October 2007: Fixed the thread safety problem, but led to performance regression that was reported in May 2009

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

Fixing a Thread-Unsafe Class in Groovy

```
1  class ExpandoMetaClass {
2    private boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.setInitialized(true);
6    }
7    synchronized void setInitialized(
         boolean b) {
8      this.initialized = b;
9    }
10   synchronized boolean isInitialized() {
11     return this.initialized;
12   }
13 }
```

```
1  class ExpandoMetaClass {
2    private volatile boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.setInitialized(true);
6    }
7    void setInitialized(boolean b) {
8      this.initialized = b;
9    }
10   boolean isInitialized() {
11     return this.initialized;
12   }
13 }
14
```

September 2009: Fixed performance regression

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

Fixing a Thread-Unsafe Class in Groovy

```
1  class ExpandoMetaClass {
2    private boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.setInitialized(true);
6    }
7    synchronized void setInitialized(
         boolean b) {
8      this.initialized = b;
9    }
10   synchronized boolean isInitialized() {
11     return this.initialized;
12   }
13 }
```

```
1  class ExpandoMetaClass {
2    private volatile boolean initialized;
3    synchronized void initialize() {
4      if (!this.initialized)
5        this.setInitialized(true);
6    }
7    void setInitialized(boolean b) {
8      this.initialized = b;
9    }
10   boolean isInitialized() {
11     return this.initialized;
12   }
13 }
14
```

Is the `synchronized` keyword required? Would `volatile` not suffice?

Difference between volatile and synchronized in Java

Real-World Performance Bugs: Apache

Patch for Apache Bug 45464	What is this bug
modules/dav/fs/repos.c status = apr_stat (fscontext->info, - APR_DEFAULT); + APR_TYPE);	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

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.

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

Real-World Performance Bugs: Mozilla 1

Mozilla Bug 66461 & Patch	What is this bug

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

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.

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

Real-World Performance Bugs: Mozilla 2

Mozilla Bug 515287 & Patch	What is this bug

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

Users reported that Firefox cost 10 times more CPU than Safari on some popular Web pages, such as `www.gmail.com`. Lengthy profiling and code investigation revealed that Firefox conducted an expensive GC process at the end of every `XMLHttpRequest`, which was too frequent. A developer then recalled that GC was added there few 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 before. This bug was fixed by removing the call to GC.

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

Real-World Performance Bugs: Mozilla 3

Mozilla Bug 490742 & Patch	What is this bug
for (i = 0; i < tabs.length; i++) { ... - tabs[i].doTransact(); } + doAggregateTransact(tabs); *nsPlacesTransactionsService.js*	*doTransact* saves one tab into `bookmark' SQLite Database. **Firefox hangs** @ `bookmark all (tabs)'. The patch adds a new API to aggregate DB transactions.

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`.

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

Real-World Performance Bugs: MySQL

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.

MySQL synchronization-library developers implemented a `fastmutex_lock` for fast locking. Unfortunately, unit tests 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.

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	Web browser	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

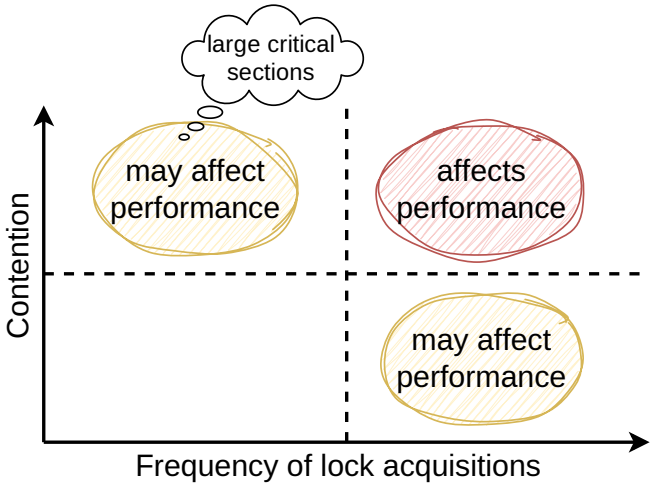
- Inefficient function call combinations (e.g., “bookmark all (tabs)”)
- Wrong API interpretation (e.g., APACHE HTTPD)
- Redundant work (e.g., MySQL `fastmutex_lock`)
  - Wrong functional implementation
- Resource contention (e.g., sub-optimal synchronization and false sharing)
  - Many synchronization 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

Spectrum of Synchronization Operations

Type	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

- Wrong granularity choice
  - For example, refine coarse locks into finer-grained locks
- Over-synchronization
  - CS data is thread-local or read-only or may write to disjoint addresses
  - Operations are already protected by another lock
- Use of improper primitives
  - For example, use of try locks in case of repeated failures, blocking synchronization with condition variables might be better
- Asymmetric contention
  - For example, 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
  - Measure thread idle time, thread synchronization time
  - Thread Profiler, IBM Lock Analyzer, SyncProf, ...
- Study impact of critical sections on the critical paths of applications
  - Focus on locks that can impact performance
- Detect load imbalance

Speculative Lock Elision

Potential Parallelism Hurt by Synchronization

```
1 LOCK(locks->error_lock);
2 if (local_error > multi->err_multi)
3   multi->err_multi = local_error;
4 UNLOCK(locks->error_lock);
```

ocean

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

Potential Parallelism Hurt by Synchronization

Thread 1

```
1 LOCK(hash_tbl.lock)
2 var = hash_tbl.lookup(X)
3 if (!var)
4   hash_tbl.add(X);
5 UNLOCK(hash_tbl.lock)
```

Thread 2

```
1 LOCK(hash_tbl.lock)
2 var = hash_tbl.lookup(Y)
3 if (!var)
4   hash_tbl.add(Y);
5 UNLOCK(hash_tbl.lock)
```

concurrent hash table

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

Problems with Conservative Locking

- Conservative synchronization leads to serialization
- 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
- Speculative execution in OOO processors are not able to restore the inherent parallelism

Speculative Lock Elision (SLE)

Insight: Locks can be elided if atomicity can be guaranteed for all memory operations within critical sections by some means

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 and execute with the lock acquired
- Atomicity is not violated if lock release is encountered, elide lock release and commit speculative state

SLE can be provided with both software and hardware support

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

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 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
    - On external invalidation to a block with access bit set
    - On 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)

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

– 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

Need for Cache Coherence

Types of Parallelism

Instruction-level Parallelism

Overlap instructions within a single thread of execution (e.g., pipelining, superscalar issue, and out-of-order execution)

Data-level Parallelism

Execute an instruction in parallel on multiple data values (e.g., vector instructions)

```
for (int i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```

Thread-level Parallelism

Concurrently execute multiple threads

Shared Memory Multiprocessor Architecture

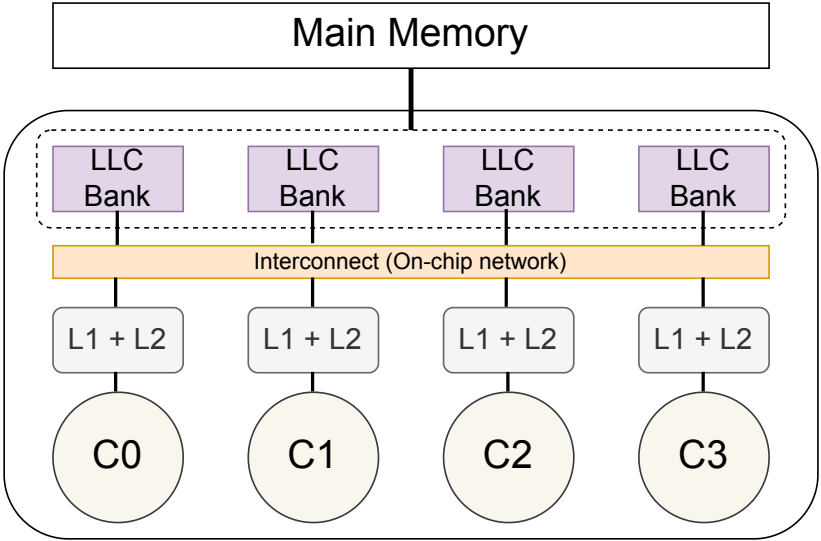
Single address space shared by multiple cores

- \+ Exploits TLP by having a number of cores
- \+ Can share data efficiently, communication is implicit through memory instructions (i.e., loads and stores)
- – Cost for accessing shared memory can be uniform or non-uniform across cores

Processors privately cache data to improve performance

Reduces average data access time and saves interconnect bandwidth

Block Diagram of a SMP



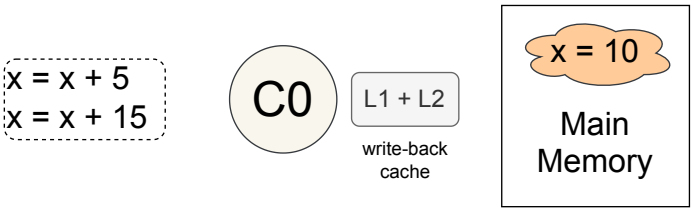
Data Coherence

Private caches create data coherence problem

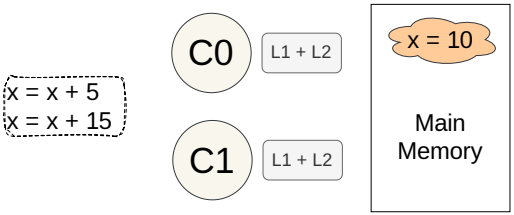
- Copies of a variable can be present in multiple caches
- Private copies of shared data must be **coherent**, i.e., all copies must have the same value (okay if the requirement holds eventually)

Consider the following sequence of operations on a single core system

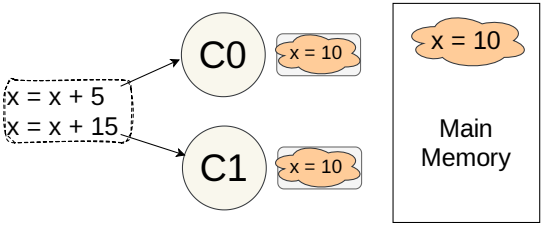
Final value of x will be 30



Coherence Challenge with Multicores

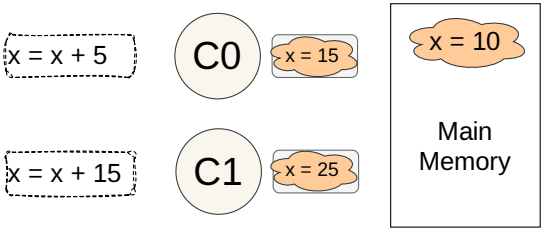


(i) Multicore system setup

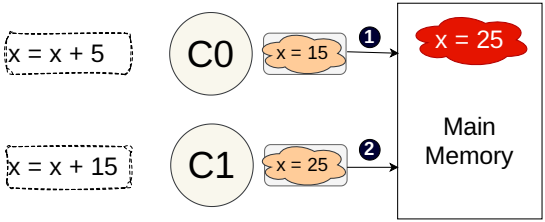


(ii) Each core reads x

Coherence Challenge with Multicores



(iii) Each core updates x in its private cache



(iv) Cores write back x, a store is lost depending on the order of write backs

Can Write-through Caches Avoid the Coherence Problem?

Assume 3 cores with write-through caches

(i) Core C0 reads x from memory, caches it, and gets the value 10

(ii) Core C1 reads x from memory, caches it, and gets the value 10

(iii) C1 writes x=20, and updates its cached and memory values

(iv) C0 reads x from its cache and gets the value 10

(v) C2 reads x from memory, caches it, and gets the value 20

(vi) C2 writes x=30, and updates its cached and memory value

Sources of Errors in the Previous Examples

Write-back cache

- Stores are not visible to memory immediately
- Order of write backs are important
- **Lesson learned**: do not allow more than one copy of a cache line in dirty state

Write-through cache

- The value in memory may be correct if the writes are correctly ordered
- Our example 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

Understanding Coherence

A memory system is coherent if the following hold:

(i) A read from a location X by a core C that follows a write by C to X always returns the value written by C provided there are no writes of X by another processor between the two accesses by C.

(ii) A read from a location X by a core C that follows a write to X by another core returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

(iii) Writes to the same location are serialized. That is, two writes to the same location by any two cores are seen in the same order by all processors.

Correctness Requirement

For sequential programs, there is only one correct output

A read from a memory location must return the “latest” value written to it

For parallel programs, there can be multiple correct outputs

- Defining “latest” precisely is crucial
- Assume that the latest value of a location is the latest value “committed” by any thread/process

Cache Coherence

Cache Coherence Protocol

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

- Essentially, makes one core's write visible to other cores by propagating the write to other caches
- Aims to make the presence of private caches functionally invisible
- Coherence protocols can operate on granularities from 1–64 bytes, usually operate on whole cache blocks (e.g., 64 bytes)

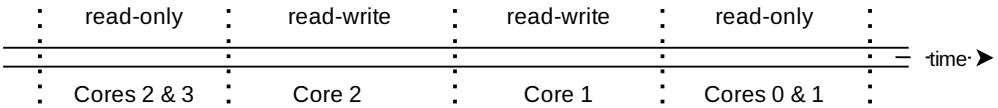
Cache Coherence Protocol Invariants

1. Enforces the Single-Writer-Multiple-Reader (SWMR) invariant

For any given memory location, at any given moment in time, there is either a single core that may write it (including read) or some number of cores that may read it

2. Data values must be propagated correctly (data invariant)

The value of a memory location at the start of a read-only time period is the same as the value of the location at the end of its last read-write time period



Alternate Definitions of Coherence

Definition 2

A coherent system must appear to execute all threads' loads and stores to a **single** memory location in a total order that respects the program order of each thread

Definition 3

A coherent system satisfies two invariants:

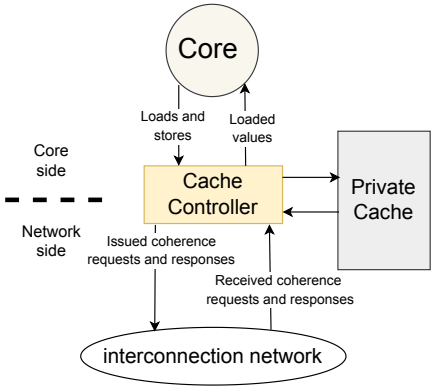
write propagation every store is eventually made visible to all cores, and

write serialization writes to the same memory location are serialized (i.e., observed in the same order by all cores)

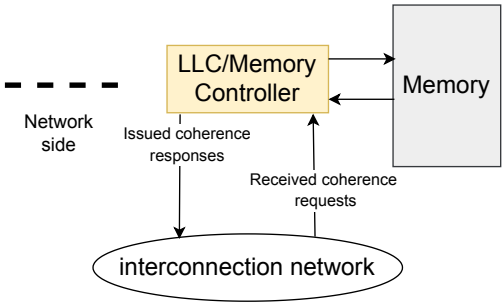
Implementing Coherence Protocols

Protocols are implemented as finite state machines called coherence controllers

A protocol formalizes the interactions between the different coherence controllers



Cache controller



Memory controller

Important Characteristics of a Cache Block

Coherence protocols are implemented by associating states with each cache block

Validity A valid block has the most up-to-date value for this block. The block may be read. It can be written if it is also exclusive.

Dirtyness A cache block is dirty if its value is the most up-to-date, and this value differs from the value in the LLC/memory.

Exclusivity A cache block is exclusive if it is the only privately cached copy of that block (i.e., the block is not cached anywhere else except perhaps in the shared LLC).

Ownership A cache or memory controller is the owner of a block if it is responsible for responding to coherence requests for that block. In most protocols, there is exactly one owner of a given block at all times.

Stable States

M, S, and I are commonly-used states

Modified (M) The block is valid, exclusive, owned, and potentially dirty. The cache has the only valid copy of the block, the cache must respond to requests for the block, and the copy of the block at the LLC/memory is potentially stale.

Shared (S) The block is valid but not exclusive, not dirty, and not owned. The cache has a read-only copy of the block. There may be multiple processors caching a line in S state.

Invalid (I) The cache either does not contain the block (not present) or it contains a potentially stale copy that it may not read or write.

Different protocol extensions add additional states (e.g., E, O, and F) to optimize for certain sharing patterns

Common Coherence Transactions

Transaction	Goal of Requestor
GetS	Obtain block in Shared (read-only) state
GetM	Obtain block in Modified (read-write) state
Upg	Upgrade block state from read-only (Shared or Owned) to read-write (Modified); Upg (unlike GetM) does not require data to be sent to requestor
PutS	Evict block in Shared state
PutE	Evict block in Exclusive state
PutO	Evict block in Owned state
PutM	Evict block in Modified state

Communication between Core and Cache Controller

Event	Response from Cache Controller
Load	If cache hit, respond with data from cache; else initiate GetS transaction
Store	If cache hit in state E or M, write data into cache; else initiate GetM or Upg transaction
Atomic RMW	If cache hit in state E or M, atomically execute RMW semantics; else initiate GetM or Upg transaction
Instruction fetch	If I-cache hit, respond with instruction from cache; else initiate GetS transaction
Read-only prefetch	If cache hit, ignore; else (optionally) initiate GetS transaction
Read-write prefetch	If cache hit in state M, ignore; else (optionally) initiate GetM or Upg transaction
Replacement	Depending on state of block, initiate PutS, PutE, PutO, or PutM transaction

Types of Coherence Protocols

Protocols differ in **when** and **how** writes are propagated

- The writes can be propagated synchronously or asynchronously
- Synchronous propagation means a write is made visible to other cores *before* returning

Two main axes to classify synchronous protocols

(i) Invalidation-based protocol and Update-based protocol

(ii) Snoopy protocol and Directory protocol

Invalidation-Based Protocols

Invalidate all cached copies before allowing a store to proceed

Need to know the location of cached copies

Solution 1 : Broadcast that a core is going to do a store and sharers invalidate themselves

Solution 2 : Keep track of the sharers and invalidate them when needed

+ Only store misses go on bus and subsequent stores to the same line are cache hits
− Sharers will miss next time they try to access the line

Update-Based Protocols

Update all cached copies with the new value of the store

- \+ Sharers continue to hit in the cache, do not need to initiate and wait for a GetS transaction to complete
- – Prevalence of spatial and temporal locality can lead to unnecessary updates, leading to increased bandwidth requirements
- – Complicates implementing many memory consistency models

Snoopy Protocol

Cache controller initiates a request for a block by broadcasting a request message to all other coherence controllers

- Each cache controller snoops (i.e., continuously monitors) the shared medium (e.g., bus or switch) for write activity concerned with its cached data addresses
- Assumes a global bus structure where communication can be seen by all
- Relies on the interconnection network to deliver the broadcast messages in a consistent order to all cores

How do you prevent simultaneous writes from different controllers?

Snoopy Protocol

Invalidate on a write

- Core that wants to write to an address grabs a bus cycle and broadcasts a “write invalidate” message
- All snooping caches invalidate their private copy of the appropriate cache line
- Core writes to its cached copy
- Any future read in other cores will now miss in cache and refetch new data

Update on a write

- Core that wants to write to an address grabs a bus cycle and broadcasts new data as it updates its own copy
- All snooping caches update their copy

Directory Protocol

Cache controller initiates a request for a block by unicasting it to the block's home memory controller

- Memory controller maintains a directory that holds state about each block in the LLC/memory (e.g., coherence state, the current owner ID, and a bitvector for the list of current sharers)
- If the LLC/memory is the owner, the memory controller completes the transaction by sending a data response to the requestor
- If a cache controller is the owner, the memory controller forwards the request to the owner cache
- When the owner cache receives the forwarded request, it completes the transaction by sending a data response to the requestor

Snoopy vs Directory Protocol

Snoopy Protocol

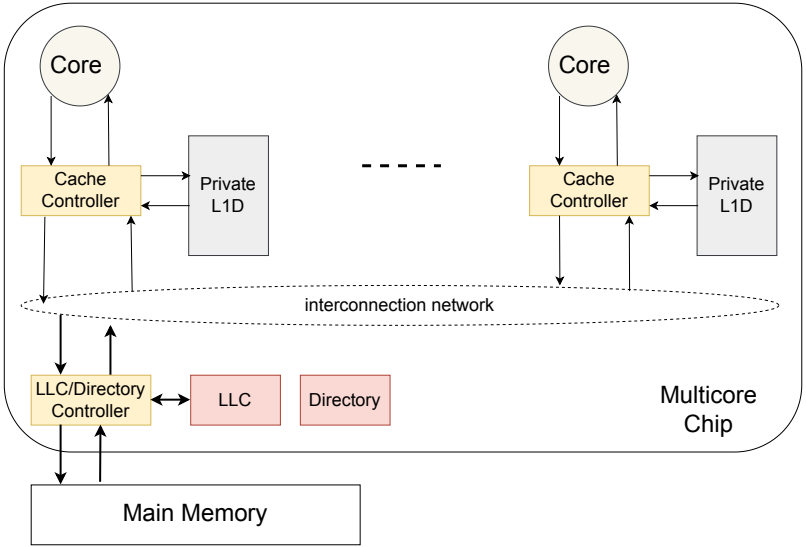
- − Does not scale to large core counts because of broadcast messages
- − Requires some ordering guarantees on messages which limits network optimizations

Directory Protocol

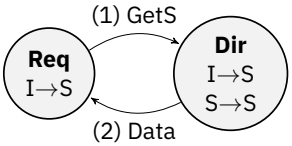
- + Scalable because messages are unicast
- + The directory can be distributed to improve scalability
- − Few transactions take more cycles when the home is not the owner
- − Memory requirement increases with core count

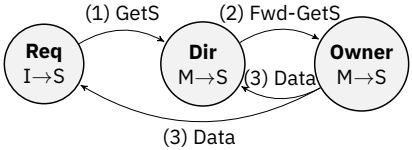
Coherence Protocols

Directory System Model

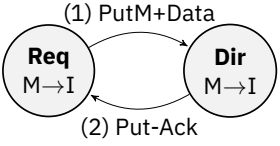


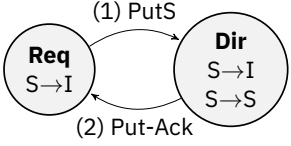
MSI Protocol





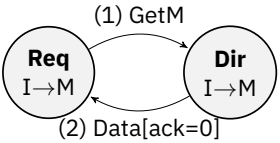
Transitions from I to S

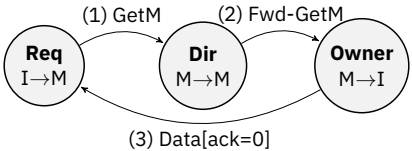


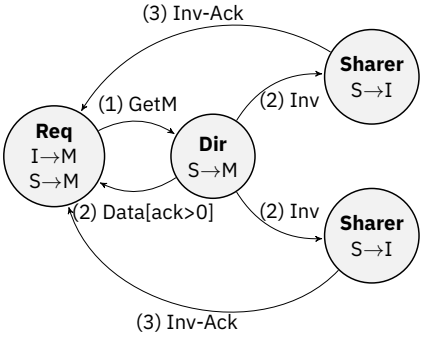


Transitions from M or S to I

MSI Protocol







Transitions from I or S to M

Usefulness of E State

Cores often read data before updating it

- Oftentimes, there is only one sharer in the system (also applicable for single-threaded programs)
- But with MSI, the core will issue two coherence transactions: GetS followed by an Upg

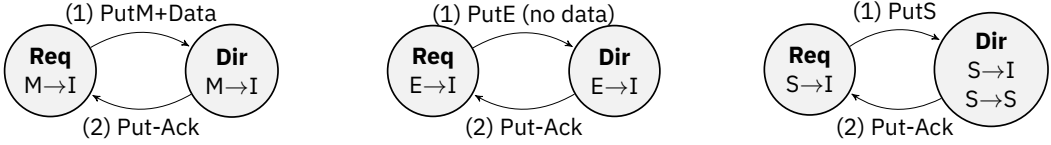
Optimization with E state

- A core issues GetS for a block
- Core gets the block in E state if there are no existing sharers
- E state indicates the cache line is clean and is the only cached copy
- The core may then silently upgrade the block from E to M without issuing another coherence request
- We will assume E is an ownership state, which implies evictions cannot be silent

MESI protocol

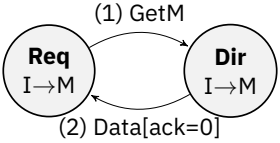
MESI Protocol

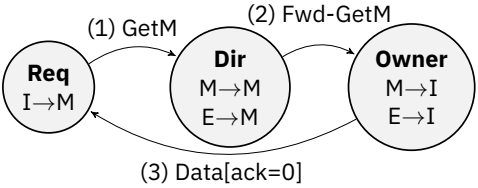
Transitions from I to S

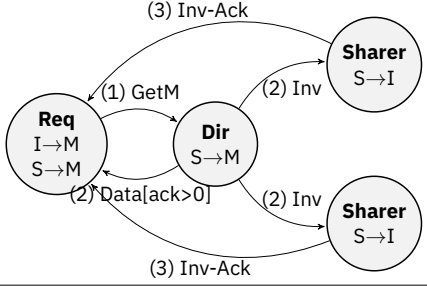


Transitions from M or E or S to I

MESI Protocol

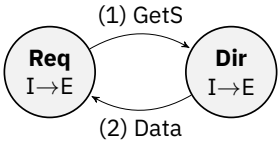






If the only sharer is the requestor, then no Inv messages are sent and the Data message from the Dir to Req has an Ack count of zero.

Transitions from I or S to M



Transitions from I to E

Cache Contention

Types of Cache Contention

Cache line contention arises from two types of read-write data sharing: true sharing and false sharing

True Sharing

- Same location is accessed by multiple cores
- Can be fixed only by means of algorithmic changes

False Sharing

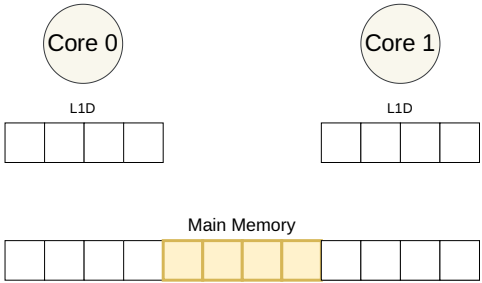
- Two unrelated locations lie on the same cache line and are accessed by multiple cores
- Can be fixed by modifying the data layout (manual or automated)

False Sharing

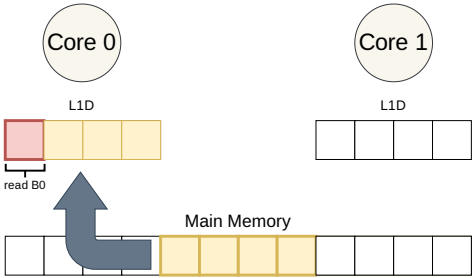
False sharing is a performance problem in cache-coherent systems

- Cores contend on cache blocks instead of data
- Can arise when threads access global or heap memory

Understanding False Sharing

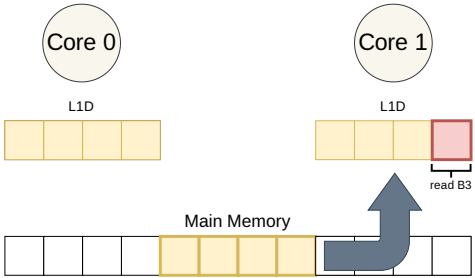


(i) Multicore system setup

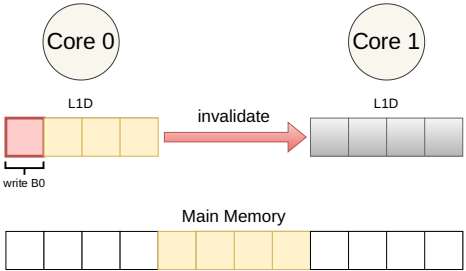


(ii) Core 0 reads block offset B0

Understanding False Sharing

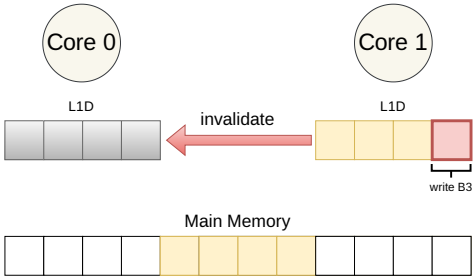


(iii) Core 1 reads block offset B3

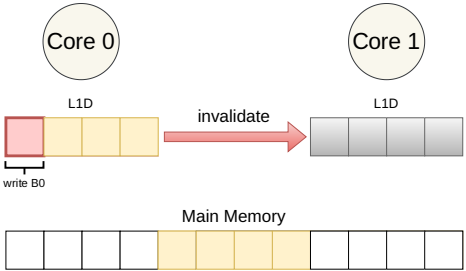


(iv) Core 0 writes block offset B0

Understanding False Sharing



(v) Core 1 writes block offset B3



(ii) Core 0 writes block offset B0

Cache misses resulting from data sharing across cores are called **coherence** misses (e.g., the write to B3 by Core 1 in (v))

Impact of False Sharing

```
1  int array[100];

3  void *func(void *param) {
     int index = *((int*)param);
5    for (int i = 0; i < 100000000; i++)
       array[index]+=1;
7  }

9  int main() {
     int first_elem = 0;
11   int bad_elem = 1;
     int good_elem = 99;
13   pthread_t thread_1;
     pthread_t thread_2;
15
     clock_gettime(CLOCK_REALTIME, ...);
17   func((void*)&first_elem);
     func((void*)&bad_elem);
19   clock_gettime(CLOCK_REALTIME, ...);
```

```
     clock_gettime(CLOCK_REALTIME, ...);
22   pthread_create(&thread_1, NULL, func,
                    (void*)&first_elem);
24   pthread_create(&thread_2, NULL, func,
                    (void*)&bad_elem);
26   pthread_join(thread_1, NULL);
     pthread_join(thread_2, NULL);
28   clock_gettime(CLOCK_REALTIME, ...);

30   clock_gettime(CLOCK_REALTIME, ...);
     pthread_create(&thread_1, NULL, func,
32                  (void*)&first_elem);
     pthread_create(&thread_2, NULL, func,
34                  (void*)&good_elem);
     pthread_join(thread_1, NULL);
36   pthread_join(thread_2, NULL);
     clock_gettime(CLOCK_REALTIME, ...);
38 }
```

https://github.com/MJjainam/falseSharing

Impact of False Sharing

```
swarnendu@vindhya:~/falseSharing$ ./a.out
array[first_element]: 300000000          array[bad_element]: 200000000          array[good_element]: 100000000


Time take with false sharing      : 330.773397 ms
Time taken without false sharing  : 173.216272 ms
Time taken in Sequential computing : 325.908369 ms
swarnendu@vindhya:~/falseSharing$ ./a.out
array[first_element]: 300000000          array[bad_element]: 200000000          array[good_element]: 100000000


Time take with false sharing      : 326.360157 ms
Time taken without false sharing  : 176.690517 ms
Time taken in Sequential computing : 324.763425 ms
```

https://github.com/MJjainam/falseSharing

Introducing False Sharing is Easy

```
1 // Global variables accessed by
2 // different threads me and you
3 me = 1;
4 you = 2;
5
6
```

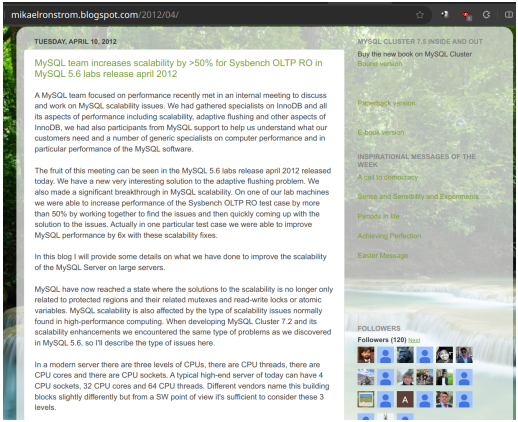
```
1 // Class/struct fields can lie on
2 // the same line
3 class X {
4   int me;
5   float you;
6 };
```

```
1 // Heap objects can lie on the
2 // same line
3 me = new Foo();
4 you = new Bar()
5
```

```
1 // Array accesses by different
2 // threads me and you
3 array[me] = 12;
4 array[you] = 13;
5
```

False Sharing in Real Applications

False sharing problems were reported in Linux kernel, JVM, and Boost library



Seeing through hardware counters: a journey to threefold performance increase

Netflix Technology Blog · Follow
Published in Netflix TechBlog · 10 min read · Nov 10, 2022

1.7K 18

By Vadim Filanovsky and Harshad Sane

In one of our previous blogposts, A Microscope on Microservices we outlined three broad domains of observability (or "levels of magnification," as we referred to them) — Fleet-wide, Microservice and Instance. We described the tools and techniques we use to gain insight within each domain. There is, however, a class of problems that requires an even stronger level of magnification going deeper down the stack to introspect CPU microarchitecture. In this blogpost we describe one such problem and

False Sharing Mitigation Techniques

- Compiler optimizations (cache block padding)
  - Inflates memory requirement, can complicate address computations
- Runtime solutions (e.g., use hardware performance counters to detect false sharing)
  - Can miss false sharing instances
- Sub-block coherence or false-sharing-aware coherence protocols
- Cache-conscious programming

Fixing False Sharing can be Non-trivial

Fixing False Sharing can be Non-trivial

Problem is often embedded inside the source code

False sharing is sensitive to

- Application behavior (e.g., mapping of threads to cores)
- Compiler toolchain (e.g., data layout optimizations and memory allocator)
  - GCC **unintentionally eliminates** false sharing in Phoenix linear_regression benchmark at certain optimization levels, while LLVM does not do so at any optimization level
- Execution environment (e.g., object placements on the cache line, hardware platform with different cache line sizes)

Detect False Sharing with `perf c2c`

Idea is to check whether loads/stores frequently hit in a remote cache line that is in M state

Input with false sharing

Compile with `gcc -O0 -g false-sharing.c -pthread`

Using `perf c2c`

```
# May need to update /proc/sys/kernel/perf_event_paranoid to -1
# sudo sh -c 'echo 1 >/proc/sys/kernel/perf_event_paranoid'
perf c2c record -F 30000 -u -- ./a.out
perf c2c report -NN -i perf.data --stdio > ./perf-report.out
# Check the analysis report
```

C2C – False Sharing Detection in Linux Perf

References I


V. Nagarajan et al. A Primer on Memory Consistency and Cache Coherence. Chapters 1,2,6–8, 2nd edition, Morgan and Claypool.


J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Sections 5.2, 5.4, 6th edition, Morgan Kaufmann.


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


T. Yu and M. Pradel. 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

References II


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


Mainak Chaudhuri. Cache Coherence. Computer Architecture Summer School, IIT Kanpur, 2018.