CS 610: Cache Coherence and False Sharing

Swarnendu Biswas

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

Sem 2024-25-I

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)

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

Thread-level Parallelism

Concurrently execute multiple threads

Shared Memory Multiprocessor Architecture

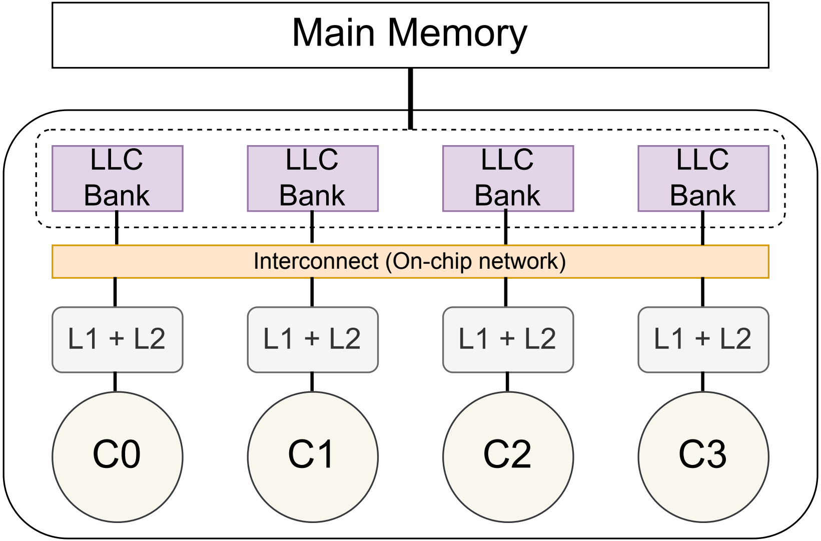
Single address space shared by multiple cores

+ 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



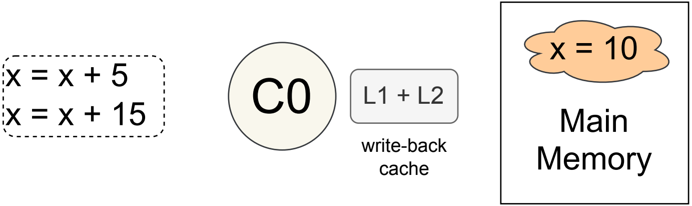
Data Coherence

Private caches create data coherence problem

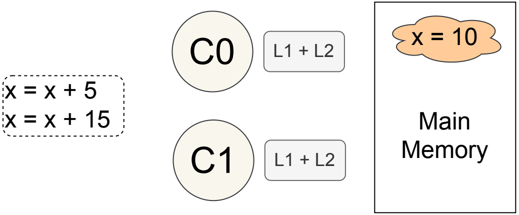
- Copies of a variable can be present in multiple caches
- Private copies of shared data must be **coherent**, i.e., all copies must have the same value (okay if the requirement holds eventually)

Consider the following sequence of operations on a single core system

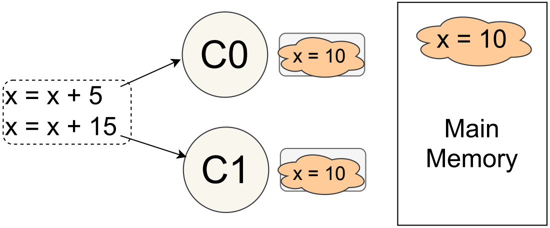
Final value of `x` will be 30



Coherence Challenge with Multicores

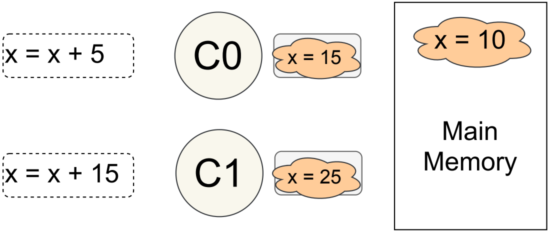


(i) Multicore system setup

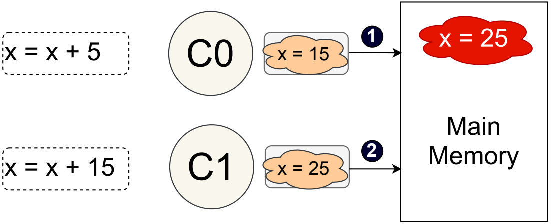


(ii) Each core reads x

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

- Core C0 reads `x` from memory, caches it, and gets the value 10
- Core C1 reads `x` from memory, caches it, and gets the value 10
- C1 writes `x=20`, and updates its cached and memory values
- C0 reads `x` from its cache and gets the value 10
- C2 reads `x` from memory, caches it, and gets the value 20
- 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

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 presence of private caches functionally invisible
- Coherence protocols can operate on granularities from 1–64 bytes, usually operates on whole cache blocks (e.g., 64 bytes)

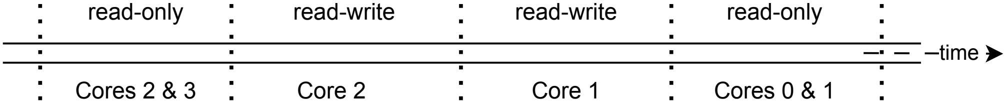
Cache Coherence Protocol Invariants

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)

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

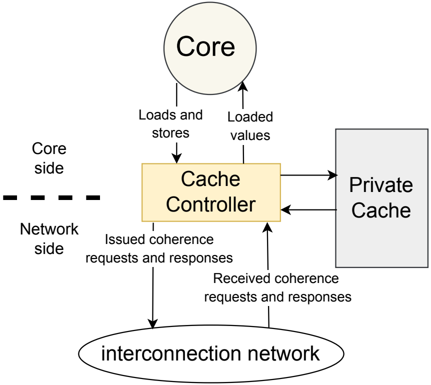
Definition 3

A coherent system satisfies two invariants: (1) write propagation: every store is eventually made visible to all cores, and (2) write serialization: writes to the same memory location are serialized (i.e., observed in the same order by all cores)

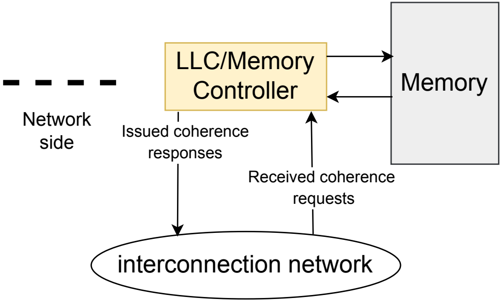
Implementing Coherence Protocols

Protocols are implemented as finite state machines called coherence controllers

The 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

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?

The problem of simultaneous writes is resolved by bus arbitration.

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

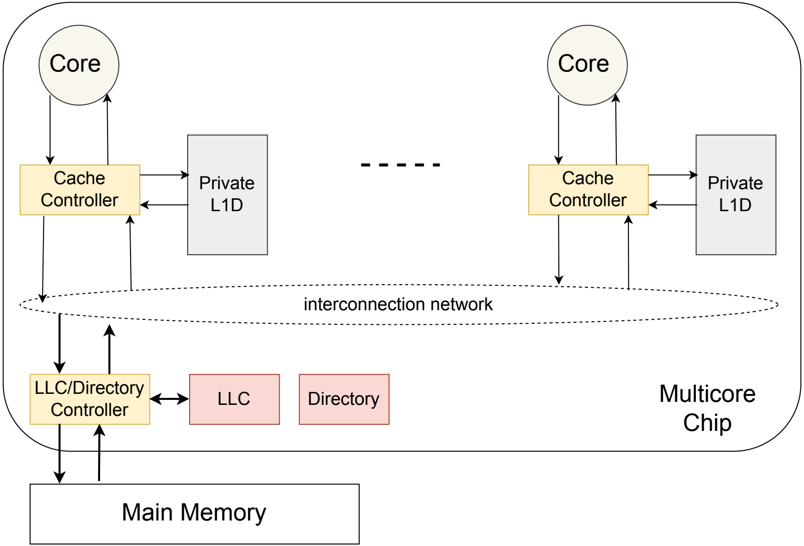
+ The directory can be distributed to improve scalability

– Few transactions take more cycles when the home is not the owner

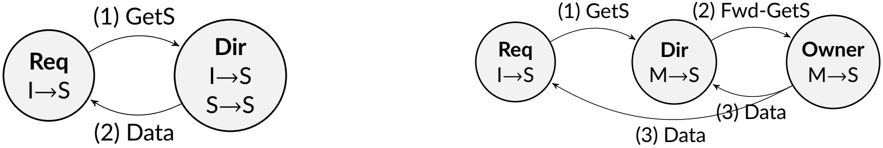
– Memory requirement increases with core count

Coherence Protocols

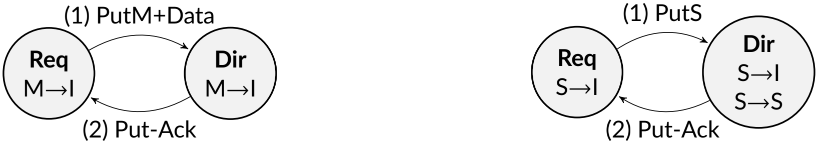
Directory System Model



MSI Protocol

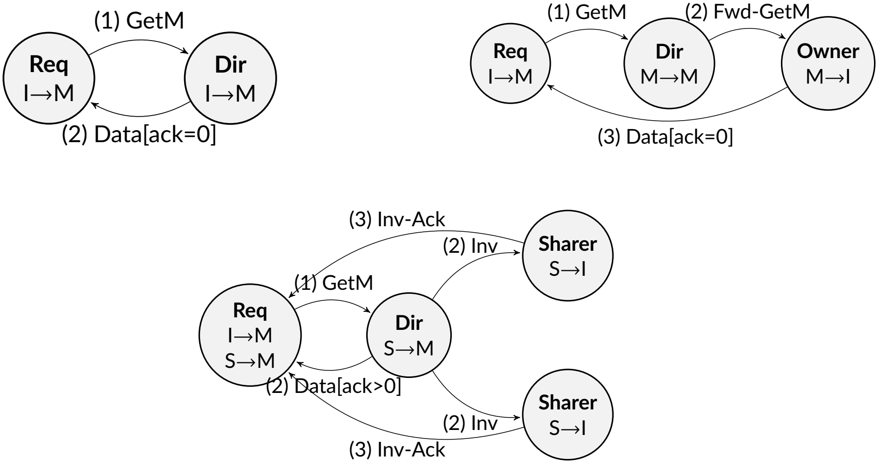


Transitions from I to S



Transitions from M or S to I

MSI Protocol



Transitions from I or S to M

Usefulness of E State

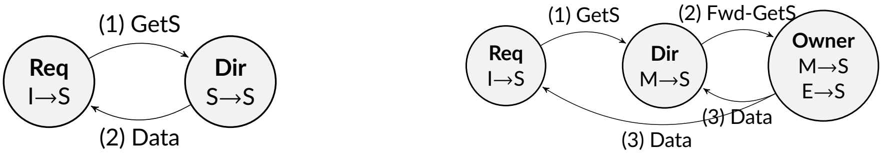
Cores often read data before updating it

- 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 a Upg

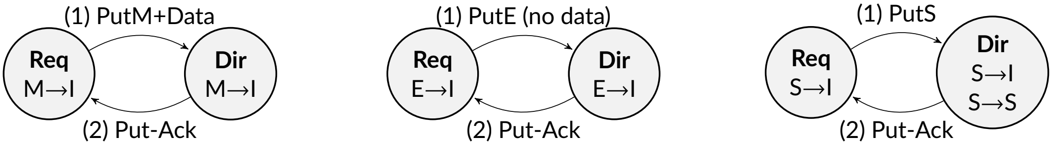
Optimization with E state

- 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

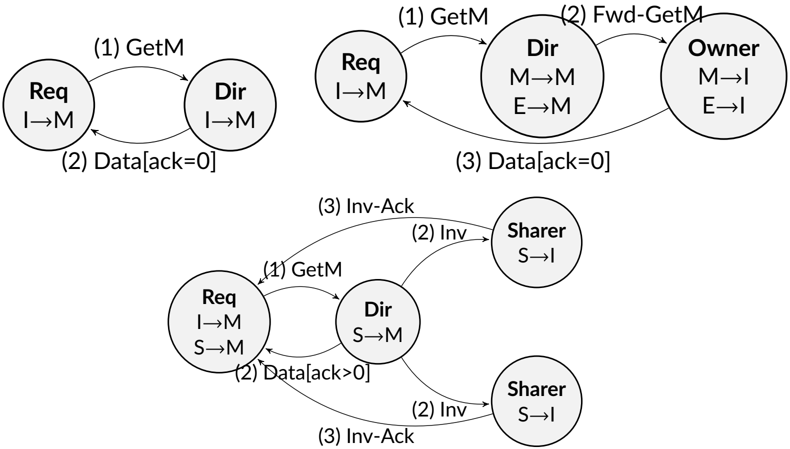


Transitions from I to S



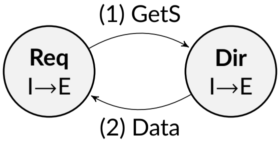
Transitions from M or E or S to I

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

Adding an Owned (O) State

Suppose a cache has a block in state M or E and receives a GetS

The cache changes the block state from M or E to S and sends the data to **both** the requestor and the memory controller.

Why is it necessary to send the data to the memory controller?

Adding an Owned (O) State

Owned state indicates that the block is valid, read-only, dirty, and the cache is the owner

Advantages of MOESI protocol (used in AMD Opteron)

- + Eliminates the extra data message to update the LLC/memory when a cache receives a GetS request in the M or E state
- + Eliminates potentially unnecessary writes to the LLC if the block is written again before being written back to the LLC
- + Allows subsequent requests to be satisfied by the private cache instead of the slower LLC or far-slower memory

Adding a Forwarding (F) State

Suppose a cache has a block in the S state and receives a GetS or GetM

- GetS is satisfied from the slow LLC/memory
- GetM is responded to by all the sharing caches, bombarding the requestor with redundant responses

Adding a Forwarding (F) State

The cache holding a block in the F state is a designated responder for any requests for the given line

- If any cache holds a line in the S state, at most one (other) cache holds it in the F state
- Memory responds in case no cache has the line in F state
- The most recent requestor of a shared line is assigned the F state. When a cache in the F state responds, it gives up the F state to the new cache to minimize chances of eviction.

Advantages of MESIF protocol (used in Intel i7)

+ Allows faster responses in case of GetS and reduces messages for GetM requests

Cache Contention

Types of Cache Contention

True Sharing

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

False Sharing

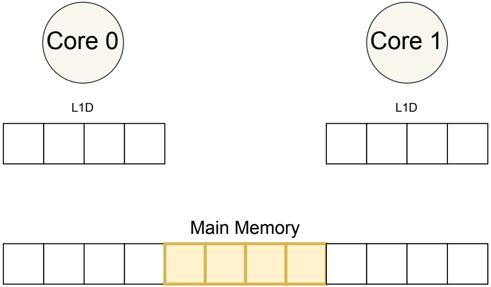
- Two unrelated locations lie on the same cache line
- Can be fixed by layout modifications (manual or automated)

False Sharing

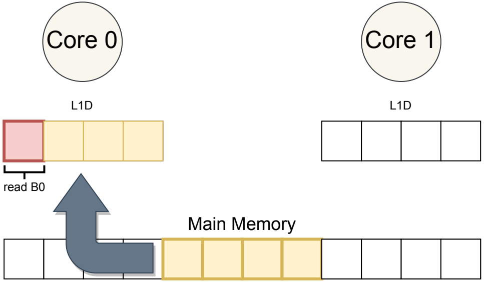
False sharing is a performance problem in cache-coherent systems

- Cores contend on cache blocks instead of data
- Can arise when threads access global or heap memory, thread-local storage and local variables can be ignored

Understanding False Sharing

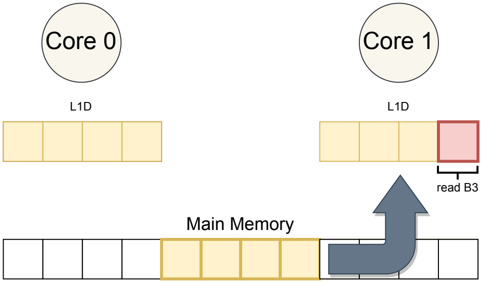


(i) Multicore system setup

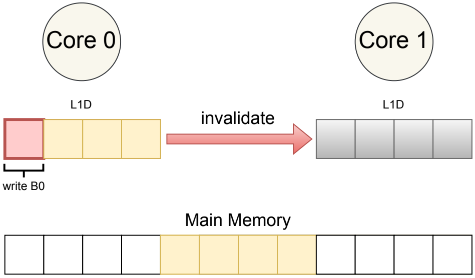


(ii) Core 0 reads block offset B0

Understanding False Sharing

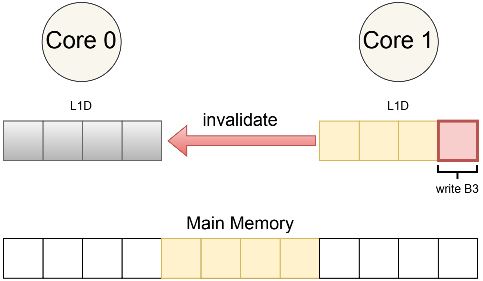


(iii) Core 1 reads block offset B3

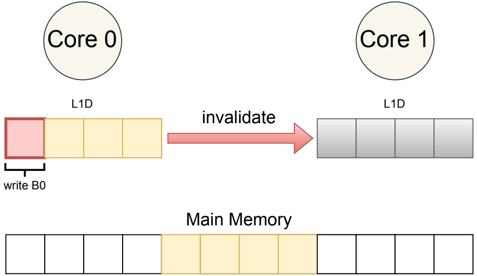


(iv) Core 0 writes block offset B0

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

```
int array[100];

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

int main() {
  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_el
  pthread_create(&thread_2, NULL, func,
                            (void*)&bad_ele
  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_el
  pthread_create(&thread_2, NULL, func,
                            (void*)&good_ele
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  clock_gettime(CLOCK_REALTIME, ...);
}
```

https://github.com/MJjainam/falseSharing

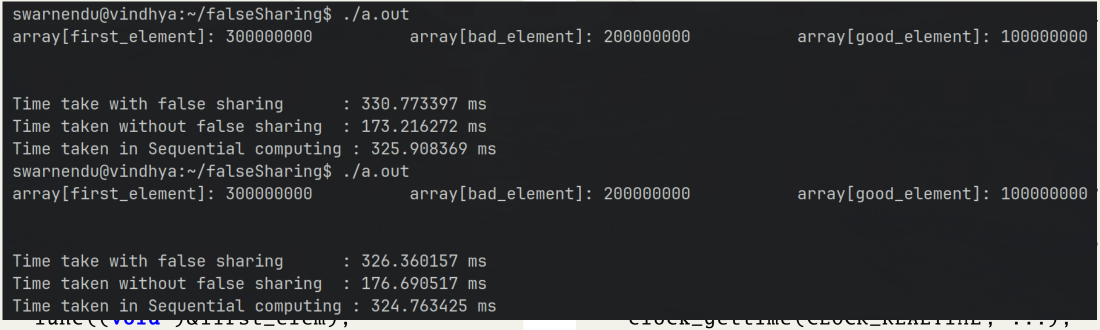
Impact of False Sharing

```
int array[100];

void *func(void *param) {
  int index = *((int*)param);
```

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

```
    clock_gettime(CLOCK_REALTIME, ...);
    pthread_create(&thread_1, NULL, func,
                                    (void*)&first_el
    pthread_create(&thread_2, NULL, func,
                                                    _elen
                                                    st_el
                                                    d_ele
    clock_gettime(CLOCK_REALTIME, ...);
}
```



```
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

```
// Global variables accessed by
// different threads me and you
me = 1;
you = 2;


```

```
// Class/struct fields can lie on
// the same line
class X {
  int me;
  float you;
};
```

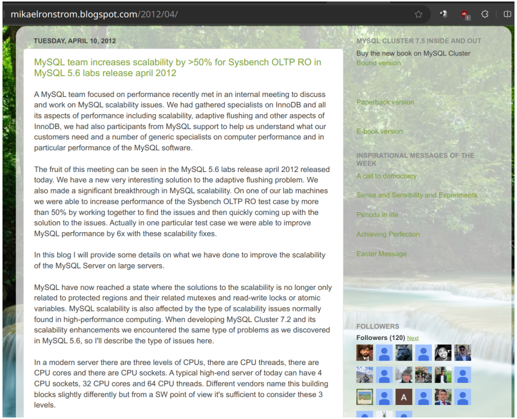
```
// Heap objects can lie on the
// same line
me = new Foo();
you = new Bar()

```

```
// Array accesses by different
// threads me and you
array[me] = 12;
array[you] = 13;
```

False Sharing in Real Applications

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



mikaelronstrom.blogspot.com/2012/04/

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 the issues. Actually in one particular test case we were able to improve MySQL performance by 6x with these scalability fixes.

In this blog I will provide some details on what we have done to improve the scalability of the MySQL Server on large servers.

MySQL have now reached a state where the solutions to the scalability is no longer only related to protected regions and their related mutexes and read-write locks or atomic variables. MySQL scalability is also affected by the type of scalability issues normally found in high-performance computing. When developing MySQL Cluster 7.2 and its scalability enhancements we encountered the same type of problems as we discovered in MySQL 5.6, so I'll describe the type of issues here.

In a modern server there are three levels of CPUs, there are CPU threads, there are CPU cores and there are CPU sockets. A typical high-end server of today can have 4 CPU sockets, 32 CPU cores and 64 CPU threads. Different vendors name this building blocks slightly differently but from a SW point of view it's sufficient to consider these 3 levels.

MYSQL CLUSTER 7.5 INSIDE AND OUT

Buy the new book on MySQL Cluster

Bound version

Paperback version

E-book version

INSPIRATIONAL MESSAGES OF THE WEEK

A call to democracy

Sense and Sensibility and Experiments

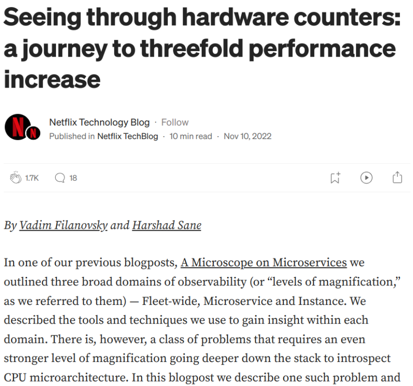
Periods in life

Achieving Perfection

Easter Message

FOLLOWERS

Followers (120) Next



Seeing through hardware counters: a journey to threefold performance increase

Netflix Technology Blog · Follow

Published in Netflix TechBlog · 10 min read · Nov 10, 2022

By Vadim Filanovsky and Harshad Sane

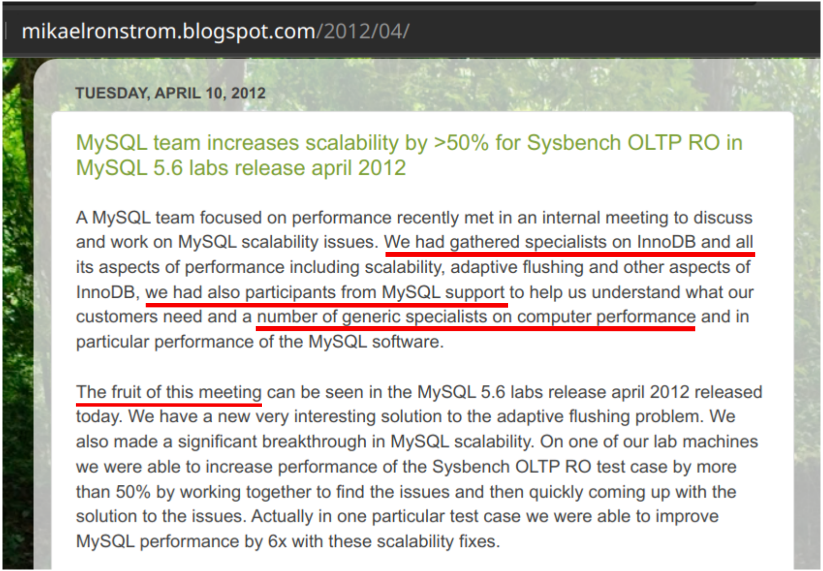
In one of our previous blogposts, A Microscope on Microservices we outlined three broad domains of observability (or "levels of magnification," as we referred to them) — Fleet-wide, Microservice and Instance. We described the tools and techniques we use to gain insight within each domain. There is, however, 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)
- Cache-conscious programming
- Runtime solutions (e.g., use hardware performance counters to detect false sharing)
- Sub-block coherence[1] or coherence at load/store granularity

[1] Snehasish Kumar et al. “Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy”. In: *MICRO*. 2012, pp. 376–388; Hongzhou Zhao et al. “Protozoa: Adaptive Granularity Cache Coherence”. In: *ISCA*. 2013, pp. 547–558.

Fixing False Sharing can be Non-trivial



mikaelronstrom.blogspot.com/2012/04/

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 the issues. Actually in one particular test case we were able to improve MySQL performance by 6x with these scalability fixes.

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., memory allocator)
- Execution environment (e.g., object placements on the cache line, 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

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

Research on False Sharing Detection and Repair

Extensively researched problem in both PL and Architecture community

Only detection	Cheetah	Liu and Liu, CGO'16
	Predator	Liu et al., PPoPP'14
	DeFT	Venkataramani et al., TACO'11
	Feather	Chabbi et al., PPoPP'18
Both detection and repair	Sheriff	Liu and Berger, OOPSLA'11
	Plastic	Nanavati et al., EuroSys'13
	Laser	Luo et al., HPCA'16
	Remix	Eizenberg et al., PLDI'16
	TMI	DeLoizer et al., Micro'17
	Huron	Khan et al., PLDI'19

References

- V. Nagarajan et al. A Primer on Memory Consistency and Cache Coherence. 2nd edition, Morgan and Claypool.
- J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. 6th edition, Morgan Kaufmann.
- A. Gupta et al. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann.
- Mainak Chaudhuri. Cache Coherence. Computer Architecture Summer School, IIT Kanpur, 2018.