Multicore Performance Demystified

Hardware matters for Java developers

Sigurt Bladt Dinesen
sidi@itu.dk

Supervisor:
Peter Sestoft

July 21, 2019

Contents

Chapter 1

Abstract

With this report, I add to the evidence that false sharing of cache lines can significantly harm the performance of practical multicore software running on managed platforms. I describe the mechanisms of hardware caches that cause false sharing overhead, and relate them to Java's synchronization constructs. I describe techniques to manipulate the memory layout of Java programs and improve multicore cache performance, all within the programming model afforded to us by Java. I verify by experiment that these techniques work, and that the performance impact they have can be considerable.

Chapter 2

Introduction

As software systems grow more complex, the tools and models we use to design them grow more abstract. It seems that the more software designers wish to accomplish in a single project, the further we move away from the bare metal of modern hardware.

High-level programming languages like Java, F#, and even C hide the mind-boggling complexities of the hardware our programs run on, in favour of simpler abstractions that make it easier (or even feasible) to design and understand complex software systems. Even with low-level languages, entire hardware components are abstracted away: In the x86-64 instruction set for example, the CPU cache is essentially invisible to the programmer, except for a few prefetch instructions. To see the scale of complexity we are talking about, consider that Intel's first i3, i5, and i7 architectures had 810 million transistors, almost three times as many as their Core 2 duo processors from just 2 years earlier [1]. Present-day Intel architectures have transistor counts in the billions. That is a lot of hardware that we do not want to have to worry about the details of.

If we accept the adage that "the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise"[2], it is easy to appreciate the work of those who hide away the complexities of computer hardware, allowing us to focus on more high-level problems in our software design. However, such abstractions can betray the unsuspecting software designer, whose understanding of the hardware they use is clouded by the abstractions they use it through.

In *A multicore performance mystery solved*[3], Sestoft brings to attention an example of such a betrayal: The curious performance impact caused by false sharing of CPU cache-lines. Using a k-means clustering implementation as example, he shows how a seemingly obvious optimization makes the program 7.7 times slower in practice.

The aim of this report is to demystify multicore performance and help eliminate false sharing overhead. We achieve this through mechanical sympathy. We see that cache-friendly design in multicore programming is radically different from cache-friendly design in single-core programming. With this understanding of cache coherence protocols, we try to bridge the gap between understanding memory hardware, and using it in Java programs. Finally, we run benchmarks of 6 different multicore programs on three different platforms to reinforce our understanding of multicore performance in Java.

The intended audience for this report is the graduate- or post-graduate level software designer who is used to working on high-level, managed platforms with garbage collection etc., and who is unaccustomed to considerations of low-level hardware details, such as cache coherence.

Because some of the source code used for the benchmarks is sensitive – it includes solutions to exercises used in the Practical Concurrent and Parallel Programming course at the IT University of Copenhagen, the source code is not published online, but is to be distributed with this report when prudent.

Chapter 3

How to read this report

It is fair to say this report has more breadth than depth. The following paragraphs should aid readers in deciding which chapters are relevant to them.

3.1 Roadmap

Readers wishing to understand the workings and impact of false sharing should read section 4.3. Readers who wish to learn techniques for avoiding false sharing overhead in java should read chapter 6. Readers wish to see evidence of false sharing overhead and the performance characteristics of data sharing between CPUs should read the experiments in chapter 9. Readers who are interested only in evidence that false sharing has a performance impact may wish to skip most of chapter 9, and focus on the uncontended- and contended- writes problems, as well as the k-means problem or the histogram problem for a more practical example.

3.2 Outline

The following outlines the chapters of this report:
Chapter 4 provides a naive description of the memory hierarchy at large. Section 4.3 goes deeper into detail with CPU caches, and ends with a description of false sharing of cache lines, based on the MESI cache coherence protocol.

Chapter 5 summarizes previous work and relevant literature on false sharing, hardware design, and multicore software design in general.

Chapter 6 explains how hardware cache coherence relates to the Java memory model, and suggests an array of techniques for manipulating the memory layout of java programs to alleviate false sharing – something Java provides little to no support for.

Chapter 7 underlines the importance of memory operations in regards to program performance, and elaborates on the differences in cache-friendly design in single- and multi-core software.

Chapter 8 describes the experimental setup and limitations of the experiments in chapter 9. Chapter 9 details multicore-performance experiments with 4 practical problems and two more contrived examples illustrating the performance impact of false sharing of cache lines.

Finally, chapter 10 summarizes the lessons learned in this project, in the form of advice to multicore programmers.

Chapter 4

Machine architecture for software designers

As software designers, we may work with different mental models of the hardware that run our programs. Particularly performance-conscious designers may be concerned with the access speeds of CPU registers vs CPU cache. C programmers may think of main memory and cache as a single, contiguous address-space that they can manipulate freely. Java programmers may ignore even the existence of the memory hierarchy altogether, and simply think about objects and their relationships.

The purpose of this chapter is to help multicore-software designers understand the workings of the memory hierarchy. To that end, I describe the memory hierarchy in a manner that is abstract enough to be useful (and understandable) for the non-hardware oriented software designer, but detailed enough to explain the performance characteristics of e.g. false sharing.

4.1 Model of the computer

We will work with a simplified model of a computer, consisting of the following components:

- CPU cores, each with their own registers. A core executes a single thread at a time, or more if it uses hyper-threading.
- Individual CPUs, each of which may contain multiple individual cores.
- Per-CPU cache hierarchies, with multiple layers of cache, some of which are shared between multiple cores.
- Main memory (colloquially RAM), shared between all CPUs.

The next page shows an image of the hardware topology of a machine with two Xeon E5 CPUs, each with 12 physical cores. The image is generaited with the Open MPI hwloc tool.


Machine (15GB total)
Package
NUMANode P#0 (7425MB)
L3 (30MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core
PU P#0
PU P#1
PU P#2
PU P#3
PU P#4
PU P#5
PU P#6
PU P#7
PU P#8
PU P#9
PU P#10
PU P#11
PU P#12
PU P#13
PU P#14
PU P#15
PU P#16
PU P#17
PU P#18
PU P#19
PU P#20
PU P#21
PU P#22
PU P#23
Package
NUMANode P#1 (8036MB)
L3 (30MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core
PU P#24
PU P#25
PU P#26
PU P#27
PU P#28
PU P#29
PU P#30
PU P#31
PU P#32
PU P#33
PU P#34
PU P#35
PU P#36
PU P#37
PU P#38
PU P#39
PU P#40
PU P#41
PU P#42
PU P#43
PU P#44
PU P#45
PU P#46
PU P#47

The image shows that main memory is shared between the two CPUs, each CPU has its own L3 cache that is shared among its cores, and each physical core has its own L2, L1d and L1i caches, shared between its virtual cores. Cache layer 1 is divided into separate caches for data (L1d) and CPU instructions (L1i).

This model excludes many aspects of real computers. Such as: The buses that transfer data between hardware components, address-translation hardware such as translation lookaside buffers, hard drives, and network interfaces. Each of these may or may not have their own interesting multicore-performance characteristics, that we shall simply ignore.

In this model, main memory is the slowest resource we work with. As we shall see in section 9, reading from main memory takes on the order of 35-140 nanoseconds, whereas reading from L1 cache can be done in 1-4 nanoseconds. It follows directly from this fact that the cache-friendliness of a program can significantly affect performance: If a program performs a billion reads, this is the difference between a second and a few minutes.

So why not built larger caches? As commented earlier, larger caches generally mean longer access times. Going to extremes, McKenney [4] notes that in order to access storage within the time of a single 5GHz CPU-clock cycle, the storage can be no more than 3 centimeters away from the CPU core, or we would have to transfer the data faster than the speed of light. Furthermore; we do not (yet) use light to transfer information between the components of a computer. We use low-voltage electricity, which is (according to [4]) about 3-30 times slower.

4.2 Main memory

The main memory functions as a kind of backbone of the memory hierarchy. It is large (often in the tens of gibibytes), and when working with high-level languages, we tend to think about the entire memory hierarchy in terms of how the main memory works: A contiguous storage space, divided into fixed chunks of equal size, each of which can be accessed individually using fixed-width addresses. This "fixed-width" is generally what is meant when we say an architecture is "x-bit": A 64-bit architecture uses memory addresses that are 64-bit long, and has 64-bit long words.

4.3 CPU caches

Regardless of how the software designer thinks of memory, the CPU cache is not generally a part of their interface: cache access happens transparently as we access the main memory. It is not however transparent with respect to performance, and cache-sympathetic software design can yield significant performance gains. A famous example of this is found in the 2012 `GoingNative` keynote by Stroustrup [5], in which he explains that insertion into linked lists is much slower than insertion into arrays/vectors, because of the unpredictable memory access pattern exhibited by traversing a linked list. In this section we take a look at the performance characteristics of CPU caches, and see that cache-friendly design in multicore programming is radically different from cache-friendly design

in single-core programming.

A word of caution: Since the cache is mostly invisible to software designers, hardware designers have a lot of freedom when designing them. This makes it difficult to model and reason about cache-behaviour across platforms. This section gives an overview of cache hardware that applies to most modern, general-purpose hardware, including x86 and x86-64 implementations. For a more comprehensive outline of modern memory-hardware, chapters 2, 3, and 6 of [6] should be of help. However, as is usually the case with hardware performance, the best way to determine how a program performs with respect to the cache is to run benchmarks on the platform it will run on.

4.3.1 What is cached and how

The cache works by storing copies of data from main memory. That way, the cache provides faster access to memory contents we expect to access in the future.

We could think of caches as general key-value stores, associating memory addresses with cached values. Such a cache could let us cache values for any set of memory locations we would like, subject only to the space constraints of the cache. This type of cache (called *fully associative*) scales poorly as it would have to store the memory addresses in addition to the cached values. Furthermore, any read or write operation must search the cache for an entry with the relevant memory address. While this may sound simple enough to software designers, cache behaviours are implemented as electronic circuitry. For these reasons, large caches – such as the multi-kibibyte L1 caches closest to the CPU cores – are not fully associative. Instead, set-associative caches are used [6] [7].

A set-associative cache is like is a hardware hash table with probing and fixed-sized buckets – or "sets". Each memory address hashes to a specific set, and each set can contain a fixed number of entries, called "ways". The hash function simply takes a fixed number of the least significant bits of the address. This eliminates the need for storing the full memory address of each cache entry: Part of the address is implied by the set used. The need to search the full cache is similarly eliminated: The hardware now only needs to search within a single set.

The storage in each way in a set is called a cache line. The cache line size can be thought of as a unit size of cache operations. The cache line size is important, because it is effectively the unit size for memory operations that do not explicitly bypass the cache! Most, if not all, of Intel's x86-64 architectures use 64-byte cache lines[8].

The downside of set-associative caches is, that unlike fully associative caches, they cannot cache an arbitrary set of memory entries: In a 2-way set-associative cache, only two entries whose addresses hash to the same set can be stored at a time. If a third entry hashes to the same set, one of the two first will be evicted from the cache, regardless of how much free space is in the other sets. Indeed it is possible for a program to use only memory addresses that hash to the same set, effectively only utilizing a small portion of the cache. Since the hash function uses the least significant bits, this can generally be avoided by keeping data close together in memory.

According to Drepper, typical CPUs in 2007 used associativity levels of up to 24 ways for L2 and larger caches, and 8 ways for L1 [6]. Intel's optimization

reference-manual [8] indicates that CPUs using their Skylake microarchitecture (from 2015) use 8 ways in L1, 4 ways in L2, and up to 16 ways in L3, so the 2007 figures appear to be current.

Several things can cause data to be stored in cache. In general, reads by a CPU core from main memory are stored in cache, under the assumption that having accessed it once, the data will likely be accessed again soon after. Similarly, the cache works as a buffer for writes to main memory: When a CPU writes a value to a memory address it first copies the full cache line into its own cache; then the relevant part of the cache line is overwritten in cache. The cache line is written back to memory (or to a cache higher up in the hierarchy) at a later time. Caches that buffer writes like this are known as write-back caches, as opposed to write-through caches where writes are immediately propagated into main memory.

The contents of main memory may also be cached by clever prefetch mechanisms that anticipate future accesses. For example, iterating over the elements of an array creates a memory access-pattern (also know as a "stride") that is easily predictable. It may help to think of such access patterns as a function $f(n) = \ldots$, where n is the number of accesses we have already made, and $f(n)$ is the next address we want to access. The stride of reading every third element from an array can then be described by the linear function $f(n) = 3n$ (ignoring the complications of array-element size, the array base-pointer, and virtual memory). The first access is to address $f(0) = 0$, the second to $f(1) = 3$, etc. Individual caches have associated prefetch-hardware that essentially performs regression-analysis of actual memory accesses in order to guess the constants of the stride function and perform reads before they are requested. Modern commodity hardware can generally predict strides that are linear functions.

Some hardware platforms (e.g. x64 and x86-64) also provide instructions for prefetching; letting software designers prefetch manually if the hardware prefetch mechanisms are unsatisfactory[6]. Similarly, there are so called "non-temporal" instructions available to read and write to main memory without the values being cached.

4.3.2 Data sharing and cache coherence

The fact that writes are buffered in the caches is our first hint that multicore programming is non-trivial. As different CPU cores store copies of data from main memory in their own caches (and in registers as well), it is possible for different cores to have different ideas of what the value stored at a certain memory address is.

Figure 4.1 shows 2 CPU cores, their registers, caches, and shared main memory. The CPUs have reached a state where there are 3 different values for the same memory location. None of which can be said to be more "correct" than the others. The state depicted in figure 4.1 can be reached e.g. by the following sequence of actions:

1. Main memory has the value 4 stored at location 1.
2. CPU1 reads location 1 from main memory, and decrements it by 1.
3. CPU0 calculates a new value (6) in a register, and is about to write it to location 1.

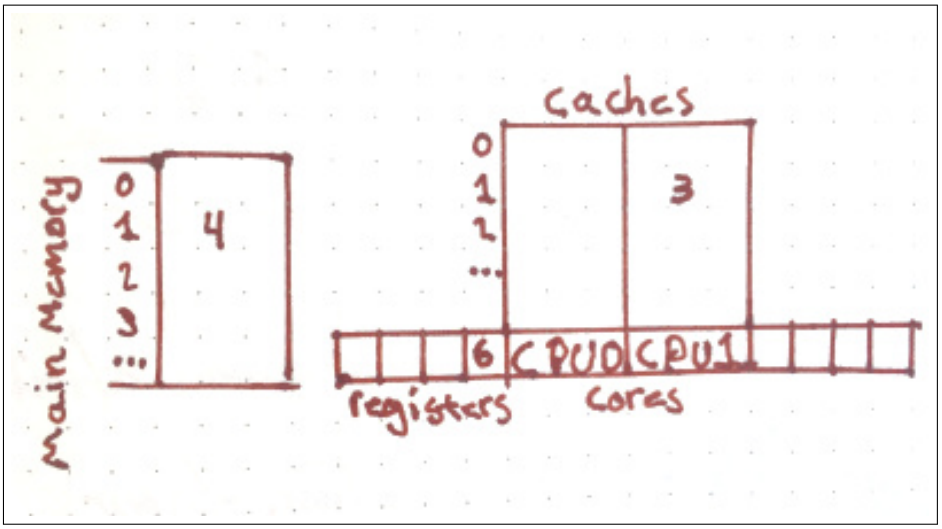


Figure 4.1: Two CPUs with incoherent views of what is stored at memory location 1

4. CPU1 writes the result (3) back to location 1, causing it to be stored in cache, but not yet in main memory.

It is up to the software designer or compiler to avoid incoherence in CPU registers. The remainder of this chapter is concerned with the how hardware helps us avoid them in cache.

Multicore architectures solve such incoherences with "cache coherence protocols". The purpose of a cache coherence protocol is to ensure that different caches agree – to some extent – what the contents of a specific cache line are. We consider the MESI protocol [9, 7], which guarantees that at most one CPU core modifies its copy of a cache line at a time.

Different architectures use different coherence protocols of varying complexity. MESI lays the foundation for the MESIF and MOESI protocols designed for Intel and AMD respectively.

The unit of the coherence protocol is called a "coherence block". On machines with hardware coherence, like x86 and x86-64, coherence blocks are cache lines. On DSM and NUMA machines, they may be pages instead[10]. In this report, I use the terms cache line and coherence block interchangeably.

The MESI protocol works by attaching one of four states to each cache line entry in each cache:

Modified means the cache line is exclusive to this cache, and has been modified by the CPU since it was cached. The cache line is invalid in the caches of all other cores.

Exclusive means the same as modified, except the cache line has not been modified since it was cached.

Shared means other caches have a copy of this cache line too.

Invalid means the cache line is not in cache. Either because it has not been cached, or because it has been evicted, or invalidated by request of another CPU.

The MESI protocol takes its name from these states.

Rather than restating the full protocol here, we will describe the most pertinent behaviours that result from read and write operations. This is illustrative of the protocol, and sufficient for our purposes.

The default behaviour, with little overhead or benefit from the coherence protocol, is as follows: When a CPU core writes to cache, it must first ensure that the cache line is in the S or M state. If it is not, the CPU core must send an invalidation message to all other CPUs. The other CPU cores must then, upon receiving the invalidation message, set the cache line state to I and respond with an acknowledge message. If one of the other CPUs has the cache line in the M state, it will send the value of the cache line along with the acknowledge message. Other CPU cores may be able to "snoop" the returned value into their caches as well, if they are connected to an interconnect between the requesting and responding cores. Read operations work the same way, but the other CPU cores set the cache line state to S instead of I.

To avoid stalling, when a core performs a write it does not actually wait for the acknowledge messages. Instead it records the write to its per-core store buffer, and continues execution. Later, when the cache line arrives in its cache, the buffered value is written into the cache. Similarly, when receiving an invalidation message, the core does not immediately invalidate the cache line. It sends back the acknowledge message – along with the cached value if the cache line is set as M – and then queues the actual invalidation in a per-core invalidation queue. The CPU core will however process the invalidation before sending any other messages regarding that cache line[7].

The guarantees provided by the coherence protocol by default are very weak. To reap the benefits of the coherence protocol, we must endure its full overhead. Software designers achieve this by using memory barrier instructions. Different sets of memory barrier instructions are provided by different operating systems and architectures. What matters to us is that some of them are used to force CPU cores to flush the store buffers and to process the invalidation queues, causing momentary coherence between the cores that execute them. Essentially, memory barriers cause the CPU cores to stall while they perform coherence operations.

It bears repeating that cache operations are always performed in accordance with the coherence protocol, regardless of whether or not we use memory barriers. This means the overhead of sending invalidation and acknowledge messages is always present. The memory barriers affect the use of store buffers and invalidation queues, thus improving coherence and increasing the coherence overhead.

4.3.3 False sharing of cache lines

If two CPU cores write to the same memory location, the coherence block or cache line containing that location is passed from one cache to the other. This is the wanted outcome, and ensures coherence between the caches. If two CPU cores write to *different* memory locations inside the same cache line, the exact same thing happens, but with no benefit. The coherence protocol dictates the

cache invalidations in order to achieve cache coherence, but there is no incoherence, as the CPU cores do not share any data, just the cache line. We use the term "false sharing of cache lines" to describe two or more CPU cores accessing disjoint sets of data located within the same cache line. The unnecessary cache invalidations caused by false sharing are called "false invalidations". We use the term "false sharing overhead" to refer to the full overhead of the coherence protocol caused by false sharing of cache lines, including sending and processing the false invalidations.

Consider the two threads in code snippet 4.1 running in parallel on two cores. The threads never operate on the same memory location, but if the two array elements are stored in the same cache line the cores will keep sending false invalidation messages to each other. Without memory barriers this causes unnecessary coherence messages to be sent back and forth, but much of the overhead will be hidden by the store buffers and invalidation queues. With memory barriers, the CPU cores will be forced to stall while they wait for each other to invalidate the cache line and send the acknowledge message back.

```
final int[] arr = ...;
...
//Thread 0
for(;;) {
        arr[0]++;
}
...
//Thread 1
for(;;) {
        arr[1]++;
}
...
```

Code snippet 4.1: Pseudo code with two threads accessing different sets of data, that may be located in the same cache line.

The table in figure 4.2 shows one possible sequence of cache operations when executing the threads. The sequence in the table assumes that the increment operations are surrounded by memory barriers, the two array elements are in the same cache line, and that CPU0 executes thread 0 and that CPU1 executes thread1.

As software designers, we cannot disable the coherence protocol, nor can we change the cache line size. To eliminate sharing overhead, we must change the way we allocate data in memory. Ideally, data segments that are primarily used by a single core should be close together, and data segments that are shared between cores, and updated, should not share cache lines with other segments – unless they are always updated together.

A word on words

The previous sections make some broad statements regarding word size and the nomenclature of "x-bit" architectures.

Sequence #	CPU #	Operation	State	
			0	1
0			I	I
1	0	Read	E	I
2	0	Write	M	I
3	1	Read	I	E
4	0	Read	S	S
5	0	Write	M	I
6	1	Read	I	E
7	1	Write	I	M
⋮				

Figure 4.2: Possible sequence of cache operations when executing the code in snippet 4.1. The values in the state column indicates the MESI state of the shared cache line in the caches of CPU0 and CPU1 respectively, after the operation in the row has been performed.

In reality, this terminology is *not* consistent and some architectures may have word sizes that are not the same as the address size. Furthermore, definitions of the terms "word" and "address" that are both precise and useful have proven elusive: Existing 64-bit architectures do not use the full 64-bit address space, and may impose requirements on how the unused bits are set. Virtual addresses, as used by programmers, are translated to physical addresses using complicated techniques that may be implemented both in electronic circuitry and in the operating system. The term "word" seems to simply mean "some useful size for data chunks on a given architecture". Useful because memory addresses, CPU instructions, CPU registers, integers, floating point numbers, and the amount of memory that is described by a single address are *typically* the size of a word, or some multiple or fraction of it. For example, x86-64 has 64-bit words and addresses, but some implementations support 80- or even 128-bit floating point operations.

The same applies to the word "byte", but "byte" is much more consistently used to mean 8 bits.

On x86 (and x86-64), an address refers to an individual byte, not a word. However, reading from an address in main memory does not necessarily mean reading just that one byte. We generally consider the word size to be the unit of operations on main memory, but when we go through the cache, the cache line size is a more suitable unit size.

Chapter 5

Background

In this chapter we look over previous work that this report builds on. We first look at work that addresses false sharing specifically, then we move on to works describing multicore hardware and multicore software design in general.

In his paper from 2017 [3], Sestoft brings to attention the curious effects of false sharing of cache-lines on a Java runtime platform. Using a k-means clustering implementation as example, he shows how a seemingly obvious optimization (loop fusion) makes the program 7.7 times slower in practice. That data-locality can affect performance is not surprising. What is surprising is that while the sequential version of the program benefits from the change, parallel versions suffer greatly. This is due to the effect known as *false sharing of cache lines* (or simply *false sharing*). He further finds that the effect of false sharing is highly significant when locking on elements in an array, even though the individual locks are uncontended. He subsequently shows that lock-striping implementations of concurrent hashmaps can suffer greatly from false sharing.

While the difference in cache-behaviour between sequential and multicore programs is confusing and unintuitive, it is not something new. In 1989, Eggers and Katz showed that CPU bus traffic as well as cache-miss rates and trends differ between sequential (they use the term "uniprocessor") programs and multicore programs [11]. They analyze cache behaviour for a small suite of programs, distinguishing between applications with high and low per-processor data locality[1]. They find that increasing the block (or cache line) size may improve performance for sequential programs and parallel programs with high per-processor data locality, but harms performance for parallel programs with low per-processor data locality. This happens because the cost of additional cache invalidations, and subsequent misses, outweighs the coherency overhead saved by performing fewer, larger cache reads. They also find that increases in cache size shifts the type of cache misses that occur in parallel programs: The larger the cache, the more cache misses occur due to the coherency protocol (both in absolute terms and relative to the total number of cache misses). The authors suggest that the problem can be mitigated either by an optimizing compiler or runtime environment, arranging shared data so that high per-processor data locality is achieved.

[1] They define programs having high per-processor data locality as programs that perform sequences of operations on a contiguous section of memory, where that section of memory is not (or rarely) accessed by more than one processor at a given time.

Like the loop fusion optimization that introduced the problem in [3], it may seem that we need to significantly change our parallel algorithms to avoid false sharing. However, it turns out that making simple transformations to the data layout can go a long way. In [3], adding padding around relevant fields significantly reduces the overhead.

In 1995, Jeremiassen and Eggers used static analysis and compile-time code transformations to drastically reduce false sharing in a small suite of coarse-grained[2] parallel programs [12] (the suite used was not the same as in [11]). Using three different code transformations, they reduced false sharing by 64% on average, and by more than 90% in the programs that were not locality-optimized by the programmers beforehand. The benefits reaped in terms of execution time varies greatly over the cache parameters of the platform, and the number of processors used. For one program, the optimizations increased the speedup gained from parallelism by a factor of ~ 2.4, yielding a ~ 7 times faster execution time than the sequential version, and ~ 3 times faster than the unoptimized parallel version.

In 1990, Torrellas et al. [13] analyzed the effects of sharing (both true and false) on a small suite of programs, before and after implementing a set of locality optimizations. Curiously, they seem to expect that existing compilers already pad around synchronization variables to prevent false sharing. As we shall see, and as found in [3], this is not the case with Java. In fact, I am not aware that this is the case for any particular compiler except for the work produced in [12]

Their work [13] shows that focusing on data sharing is particularly important when using optimizing compilers. Not because optimizing compilers make sharing worse (though they can), but because they are good at eliminating memory accesses to processor-private data e.g. by keeping data in CPU registers. Since they cannot do the same with shared data, as it would circumvent the coherence ensured by the cache, memory access to shared data often dominates IO traffic for parallel programs. Torrellas et al. [13] provide two sets of optimizations: One requiring detailed profiling information about the program to be optimized, and one that requires no such information. In the authors' own words, their optimizations had a "small but significant impact". Across their experiments, cache misses are reduced by 0.2-24%. One might speculate that the comparatively modest reductions are due to the small cache line size used in their simulations (4-16 bytes vs. 4-256 bytes in [12], and 64 bytes in [3]).

For the purposes of this report, false sharing is defined as CPUs accessing disjoint data sets placed the same cache-coherence block – or cache line – as explained in chapter 4. We gauge the impact of false sharing by hand-optimizing programs to avoid it, and benchmarking the programs with and without those optimizations. Measuring false sharing impact in this way will provide an approximation at best. Unsatisfied with this imprecision, some authors have tried to bridge the gap between defining false sharing, and defining the impact of false sharing:

In 1993, Bolosky and Scott [10] considered a handful of definitions of false sharing, and conclude that none of them are satisfying. They wish to find a definition that:

- Agrees with intuition in that it has has numerical value corresponding to

[2]Coarse-grained here refers to the granularity of of parallelism, not data sharing.

the cost savings attained by eliminating all false sharing. Hence it never grows as data is split over coherence blocks.

- Allows the properties of false sharing to be stated and proven as mathematical theorems.
- Is practically measurable for real programs.

It seems unlikely that any definition of false sharing will satisfy their criteria: Definitions can be based on comparing a program's behaviour with that of an idealized version[3], or on analyzing memory-operation sequences to assess unnecessary communication. In either case, the definition must be able to distinguish the performance characteristics of false sharing from those of all other communication in the memory hierarchy. Otherwise it must accept oddities like negative costs of false sharing, as optimizations to eliminate false sharing will often incur different IO overheads. No obvious solution for distinguishing the performance impacts in the memory hierarchy presents itself, and negative values for false sharing is unacceptable to Bolosky and Scott.

While a definition satisfying the above criteria would be helpful, we – as software designers – do not need it in order to understand how false sharing occurs. We certainly do not need it in order to avoid the costs associated with false sharing. In fact, the "intuition" requirement directly contradicts our goal: Avoiding performance pitfalls. That is, we wish to improve the execution time of our programs where possible, so if an optimization increases execution time in spite of reducing false sharing, we do not consider it an optimization.

In his paper from 2010 [7], Mckenney describes cache coherence behaviours and memory barriers using detailed examples. The focus is on the MESI cache coherence protocol, initially presented, but apparently not named, by Papamarcos and Patel [9]. Mckenney's paper does not explicitly deal with false sharing, but lays the foundation for understanding it from a hardware perspective. In his book, *Is Parallel Programming Hard, And, If So, What Can You Do About It?*[4], McKenney goes into much more detail with the design of CPUs as well as the memory hierarchy. Unlike the 2010 paper, the focus of the book is software design, and several design patterns for multicore software are discussed, in addition to formal verification methods for multicore software. The book mentions and describes false sharing, but for the most part assumes the reader will know how to avoid it (the target audience appears to be software designers experienced with sequential C programming).

In *What Every Programmer Should Know About Memory* [6], Drepper focuses on both modern and historical designs used in the memory hierarchy. Drepper describes different types of caches, in enough detail to understand the performance characteristics of false sharing of cache lines, as well as other unfortunate memory access patterns.

An authoritative work on hardware architecture, *Computer Architecture: A Quantitative Approach* [14] describes false sharing. The fourth, fifth and sixth editions go a little further into analysing false sharing overhead, but the book does not go into detail on how to avoid it.

[3] In their definitions, the authors alternate between considering idealized hardware, avoiding false sharing by using small coherence-blocks, or a policy – software or hardware based – that can place data ideally in the memory hierarchy.

Concurrent Programming in Java: Design Principles and Patterns by Lea describes multicore programming in Java. Lea describes the Java concurrency primitives available to the programmer, and the visibility and ordering effects that are guaranteed by the Java language specification. He briefly describes how object oriented design differs when writing sequential vs. multicore programs, and suggests several patterns for thread safe object oriented design.

Chapter 6

Java and memory

As previously noted, the way programming languages model memory is very different from how it is implemented in hardware. This is especially true for high-level languages, and doubly so for managed platforms like Java.

Java programmers cannot directly control things like memory barriers and memory layout. Instead we get the effect of barriers through judicious use of the `synchronized` and `volatile` keywords, the classes in the `java.util.concurrent` package, and the guarantees given by the Java Language Specification[16]. The language specification gives no such guarantees regarding memory layout, making it is harder to control: We can take steps to affect how data is laid out in memory, and we can verify the effectiveness of our efforts by benchmarking our programs, but we do not have guarantees that our optimizations will work on other Java runtimes, or with future Java versions.

In the following sections we first look at the relationship between Java's concurrency constructs and the memory hardware as described in chapter 4, then we look at 7 techniques for manipulating the memory layout in Java.

6.1 Data sharing

In Java, the concept of *happens-before order* most closely captures the purpose of memory barriers, in that it guarantees that memory operations are visible to relevant threads/CPU-cores, and that they appear to happen in the expected order.

The way we interact with the cache coherence protocol in the Java memory model does not correspond directly to the hardware view we have from [7]: From this hardware oriented point of view, memory barriers are invoked specifically to flush store buffers and invalidation queues, ensuring coherence with all other cores that do or have done the same. In Java, happens-before order is not something we ensure with an instruction or method, rather it is guaranteed when we interact in certain ways with synchronization variables like locks and `volatile` fields. One could say that in Java visibility is something we "get" rather than something we "do". This high-level and non-imperative style of incorporating multicore coherence into an object-oriented language is rather elegant, but it makes it easy to overlook concurrency constructs in code: Removing code that causes a seemingly unnecessary read can have important consequences if the

read was to a `volatile` field. It also forces software designers to use strange looking constructs, such as wrapping primitive types in seemingly useless wrapper classes, just so their fields can be declared `volatile`.

```
public static class Holder {
  public volatile int value;
}
```

Code snippet 6.1

Java provides no mechanism for guaranteeing visibility of updates to array elements. Declaring the array `volatile` will only affect the array field itself, not its contents. An array of instances of the `Holder` class in code snippet 6.1 can be used to effectively make an array of volatile elements: Instead of replacing the elements of the array, we confine ourselves to updating the `volatile value` fields of the `Holder` objects. Of course, we still need to ensure that that the initialization of the array elements is made visible. The downside of this technique, besides the silly wrapper class, is that it requires more memory. Each `Holder` instance takes up 16 bytes of memory instead of the 4 bytes an `int` would have used (we will see evidence for these numbers in the next section). There is also the additional memory overhead of storing the object pointers in the array. Additionally, there is the performance cost of the added indirect, as accessing the elements now requires reading the object pointer from the array, and *then* reading the object.

The implementers of the Java Class Library seem to favour the undocumented `sun.misc.Unsafe` class for this kind of synchronization difficulty. With regard to whether the `Unsafe` class should be used, I will let the name of the class and the fact that it is an unpublished, undocumented part of the api speak for itself. For readers who are undeterred by such details, Doug Lea's implementations of the `AtomicInteger` [17], `AtomicIntegerArray` [18], `LongAdder` [19], and `Striped64` [20] classes are well commented, highly illustrative, and well suited for learning about both the `Unsafe` class and multicore software design in general.

To facilitate mutual exclusion, Java provides locking mechanisms through the `synchronized` keyword, and through the `java.util.concurrent.locks` package. Lock implementations vary with the underlying hardware, and are usually provided by the operating system. The details therefore depend on the runtime platform, but the use of memory barriers and compare-and-swap appears to be ubiquitous. This means lock operations depend on the cache coherence protocol, which makes them subject to false sharing overhead. It is not immediately obvious that the actual lock data is stored in the object headers on the heap: The Java virtual machine could in theory keep a global table of locks. In [3] Sestoft found that spacing the allocation of locks significantly improves performance in a multicore context, leading us to believe that lock data is indeed stored in the object headers of the lock object. This means we can prevent false sharing of locks if we can avoid dense allocation of locks on the heap.

Like the mechanisms provided to us, the guarantees provided by the Java memory model do not correspond directly to those we know from hardware.

Reading from a `volatile` field ensures the visibility of updates made by other threads *before they wrote to the same field*. With what we know from [7], it is not clear why threads reading a *different* `volatile` field do not get the same guarantee.

The remainder of this report will nonetheless show that the hardware view taken in [7] is very useful when considering the multicore performance of Java programs. For this reason, with respect to multicore performance, we adopt the view that whenever the Java memory model guarantees happens-before-order, it does so through the use of memory barriers. This implies that e.g. accessing `volatile` fields, or taking or releasing a lock, causes the CPUs invalidation queues and store buffers to be flushed. Hence the cost of the coherence protocol is vastly increased for operations that guarantee happens-before-order.

6.2 Memory layout

Almost no promises are made regarding memory layout in Java. It appears to be true that objects as well as arrays are stored as contiguous memory segments, but this is entirely at the discretion of the implementers of the Java virtual machine in use. It seems that the only guarantees are that instance fields, static fields, and array *elements* are stored on the heap, and that local variables, formal method parameters, and exception handler parameters are not[16, chapter 17][21, chapter 2]. The Java memory model is only concerned with data allocated on the heap, as stack memory is not shared between threads. In fact, the Java Virtual Machine Specification[21] says very little about how the heap must be defined, except that it is shared between threads.

6.2.1 Padding techniques

The lack of guarantees regarding memory layout does not mean that all hope is lost. In the remainder of this chapter we look at ways to manipulate the Java runtime into adapting the memory layout we want. We will do so armed only with educated guesswork about how arrays and objects are allocated on the heap, and the OpenJDK's JOL tool (Java Object Layout).

The JOL tool can be found at `http://openjdk.java.net/projects/code-tools/jol/`

Spacing array elements

Let us first consider an array of primitive integers:

```
int[] arr = new int[n];
```

Code snippet 6.2

While it is not guaranteed, it is reasonable to assume that the elements of such an array will be stored in a contiguous memory segment on the heap. Indeed, experiments with locks in [3], as well as our own experiments in chapter 9, seem to confirm this assumption. This means array elements of primitive

types are prime candidates for false sharing: Java `int`s are 4 bytes, so there will be $64/4 = 16$ elements per 64-byte cache line. If different array elements inside the same cache line are accessed by different threads, we get false sharing overhead as described in chapter 4. For such use cases, it makes sense to spread the elements out in memory, by leaving unused "padding" between them. We will refer to this as a "spaced" or "padded" layout, as opposed to the normal "dense" layout.

```
//allocation
int[] arr = new int[n + n * p];
//access
int i = ...; // index if no padding were used
arr[i + (i+1) * p] = ...;
```

Padding technique A: Spaced allocation of array elements of a primitive type

Padding technique A allows us to introduce padding in arrays of primitive types. To have n useful array elements, each with p padding elements before it, we declare an array of length $n + n * p$. We then find the i'th useful element at index $i + (i + 1) * p$. We elect to add padding before, rather than after, each element under the assumption that there is shared information, such as the array length, stored immediately *before* the first array element. This may not actually be the case.

Padding between objects

Arrays of object types are a little more complicated. If we wish to prevent objects in an array from sharing cache lines – e.g. when using a `volatile` wrapper like the `Holder` class – padding technique A is insufficient. Unlike arrays of primitive types, it appears that arrays of object types do not contain the actual objects. Rather, they store an object pointer for each array element, pointing to the actual object on the heap. This behaviour is not specified in the standards, but it is a sensible implementation: Objects of the same supertype may not use the same amount of heap space. Computing the address of an array element from the base pointer and index would therefore be impractical, were the objects stored directly in the array.

To achieve a spaced memory layout with arrays of object types, we must pad the *object allocation* itself.

Padding technique B lets us allocate object types with padding between them. In addition to the objects, an array of pointers to the objects is allocated using the idea from technique A. The variables `pa` and `pb` represent the number of unused array elements we want before each useful element, and the number of times we wish to run the allocation loop, respectively. Assuming that an object pointer is 4 bytes long and a singleton `int` array is 8 bytes (the `int` itself plus an int `int` to store the array length), $64/4 = 16$ and $64/8 = 8$ are good values for `pa` and `pb` when cache lines are 64 bytes. We may need to keep references to the dummy arrays somewhere. Otherwise the garbage collector may remove them and compact the objects into the space occupied by the arrays. Figure 6.1 depicts the memory layout achieved with technique B. The figure makes

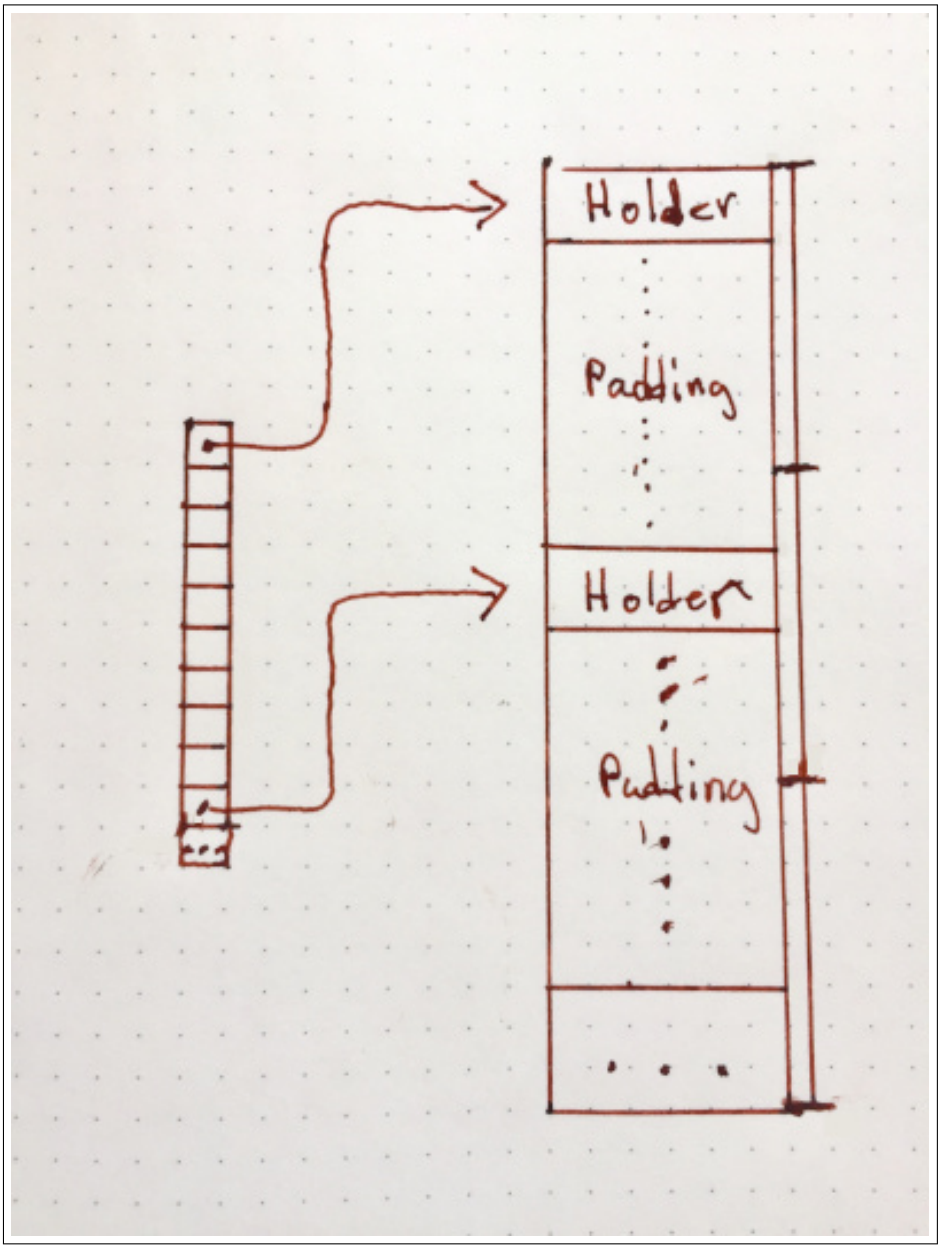


Figure 6.1: The memory layout achieved with padding technique B. Each dotted row is 8 bytes. To the left we see the array of object pointers, with 64 unused bytes between two pointers. To the right we see the `Holder` instances on the heap. Each instance occupies 16 bytes of space, followed by 64 bytes of unused `int` arrays. The markings on the right side indicate cache line boundaries. The division between the array on the left and the objects on the right is stylistic: They are both allocated on the heap.

```
Holder[] arr = new Holder[n + n * pa]
for (int i = 0; i < n; i++) {
  //padding allocation
  for(int j=0; j <= pb; j++){
    int[] dummy = new int[1];
  }
  //useful allocation
  arr[i + (i+1) * pa] = ...;
}
```

Padding technique B: Spaced allocation of an object type on the heap, and of the array elements pointing to the instances.

it clear that we are using more padding than necessary, causing the `Holder` instances to shift around in the cache lines. We could fix this by subtracting the size of a `Holder` from the amount of padding we use, but it would make it harder to write generalised methods for padding. The same applies to spacing the elements in the array.

Padding both the object allocation and the array elements means we can update both array elements and object fields without risk of false sharing. In the code, we use `int` arrays as padding on the heap. We can use anything, but it is prudent to use something with a known size. To this end arrays or plain `Object` instances are useful; as we can make slightly less uneducated guesses about their sizes, than we can for arbitrary object types.

Array indexing with indirects

```
public static Pair<int[],int[]> paddingIndirect(
                                  int n, int p) {
  int[] arr = new int[n + n * p];
  int[] indirect = new int[n];
  for(int i = 0; i < n; ++i) {
    indirect[i] = i + (i + 1) * p;
  }
  return new Pair<int[],int[]>(indirect, arr);
}
```

Padding technique C: Defining an array of indirected indices to facilitate spaced allocation of array elements of primitive types

Padding technique C gives us a different way to work with padded arrays. It works the same way as padding technique A, but instead of computing the indices of useful elements each time we need them, we store them in an indirection array. This simplifies code using the padded array, as the i'th useful element is now accessed as `arr[indirect[i]]`. This technique has the disadvantage of needing extra memory to store the indirection array. Array accesses may also

get considerably slower, as the extra memory read is likely much more expensive than computing the index was. Depending on the access pattern, the cost of the extra memory read may be hidden by cache prefetch mechanisms.

In addition to the better readability, I prefer this padding technique for many of our experiments because it might introduce less noise. Certain amounts of padding, such as 0 or a power of two, could be optimized by the compiler or hardware. The cost of the extra memory access is less likely to vary with the amount of padding.

In the same way we expanded padding technique A to work for object types, which lead us to technique B, we can expand technique C to work with object types by spacing the object allocation on the heap. To avoid repetition, we will skip the code for this, and simply refer to it at as padding technique D.

Padding within objects

Instead of having padding between objects, we can write classes that include padding inside the object instances. This has two advantages: Having the padding inside the instances is more resistant to compaction by the garbage collector, as the padding will be moved along with the object instance. It also makes code using the objects clearer by abstracting the use of padding into the class definition. On the other hand, we may still need to pad data structures used to contain the instances, and having unused fields in class definitions will not necessarily be clearer than using padding when allocating the instances.

To leverage object field layout for padding, we need to understand how objects are laid out in memory. To this end, we rely on the aforementioned JOL tool. It is tempting to simply disassemble the class files with javap, but the runtime's class loader is free to reorder object fields, so the class files tell us very little about memory layout. JOL works by instantiating classes and analysing them at runtime, allowing us to see how they are actually laid out.

Running JOL's internals command against `java.lang.Object` gives the output shown in figure 6.2.

An additional VALUE column has been omitted to save space.

There is a lot of useful information in the output. Let us dissect some of it before we move on to padding the objects. The first line tells us that we are running the HotSpot JVM on a 64-bit machine. The second line tells us that ordinary object pointers (oop) are compressed, with 0-bit shift. Compression and shifting are techniques used by the JVM to save space. The details of these techniques are not pertinent to our work, so we will suffice it to say that object pointers are compressed, which in our case means they are 4 bytes long (we will see this later). The fourth line tells us that objects are 8 byte aligned. This means that objects are stored at memory locations that are multiples of 8 bytes, hence any objects size is effectively a multiple of 8 bytes. If they are not, unused bytes will be left between objects to achieve 8 byte alignment. Lines 11-16 show the layout of the object fields, and lines 17-19 summarize the object's memory footprint. We can see from the output that an `Object` instance takes up 16 bytes on the heap, of which 12 are the object headers, and 4 are padding added so that the next item on the heap will be 8 byte aligned. This is why instances of the `Holder` class from earlier do not need more space than `Object` instances: The `int` field fills out the 4 bytes used for 8 byte alignment for `Object`.

Now that we have familiarized ourselves with JOL's output, let us take a

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

Instantiated the sample instance via default constructor.

java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0     4        (object header)
      4     4        (object header)
      8     4        (object header)
     12     4        (loss due to the next object
alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4
bytes total
```

Figure 6.2

look at the `NormalCluster` class – our version of the unpadded `Cluster` class from [3].

```
public static class NormalCluster implements Cluster {
  private volatile Point mean;
  private double sumx, sumy;
  private int count;
  ...
}
```

Code snippet 6.3: The fields of the `NormalCluster` class

Code snippet 6.3 shows the field declarations of the `NormalCluster` class. All of our cluster implementations contain these fields. They only differ by how many unused padding fields they contain.

As is made clear in [3], and as will be reiterated in chapter 9, we want to make sure that different `Cluster` instances do not share cache lines. Furthermore, we would ideally like to have the `count`, `sumx`, and `sumy` fields together in a single cache line, and the `mean` field in it's own cache line, separate from the other fields.

The output from JOL, shown in figure 6.3, shows us that all four fields are stored in a contiguous memory section of 28 bytes, followed by 4 bytes for alignment. Accounting for 8 byte alignment and object headers this means that, in the worst case, we can have all the fields of one `Cluster` instance plus the `count`, `sumx`, and `sumy` fields of another instance in the same 64-byte cache

```
com.kmeans.Clusters$NormalCluster object internals:
 OFFSET  SIZE               TYPE DESCRIPTION
      0     4                    (object header)
      4     4                    (object header)
      8     4                    (object header)
     12     4                int NormalCluster.count
     16     8             double NormalCluster.sumx
     24     8             double NormalCluster.sumy
     32     4   com.kmeans.Point NormalCluster.mean
     36     4                    (loss due to the next
object alignment)
Instance size: 40 bytes
Space losses: 0 bytes internal + 4 bytes external = 4
bytes total
```

Figure 6.3: JOL output for the unpadded `NormalCluster` class

line.

It seems that Java runtimes in general follow the same strategy for laying out object fields. Assuming 8 byte alignment: Fields are arranged in descending order of size. The sort appears to be stable, so the field order in the source code matters to some extent. If the field segment is not 8 byte aligned – due to the size of the object headers not being a multiple of 8 bytes – a 4 byte field is moved ahead of the other fields if one exists. That way, the rest of the fields are 8 byte aligned. We see this in the JOL output for the `NormalCluster` class, where `count` is moved to before the `sumx` field.

We can abuse this knowledge not only to pad objects, but to pad between fields as well.

```
  public static class PaddedCluster implements Cluster {
    private volatile Point mean;
    private int dummy1, dummy2, dummy3, dummy4, dummy5,
                dummy6, dummy7, dummy8, dummy9, dummy10,
                dummy11, dummy12, dummy13, dummy14
                dummy15, dummy16, dummy17;
    private double dDummy1, dDummy2, dDummy3, dDummy4,
                   dDummy5, dDummy6, dDummy7, dDummy8;
    private double sumx, sumy;
    private long count;
    ...
  }
```

Padding technique E: Manipulating object field layout with unused fields.

Padding strategy E lets us accheive the desired layout of the `Clusters`. The jol output for this implementation can be seen in figure 6.4. We add 8 8-byte `doubles` to clear out a cache line before the fields, 16 4-byte `ints` to separate

the `mean` field from the others, and an additional 4-byte `int` because one of them gets moved up to improve alignment.

Padding with inheritance

```
public class PaddedRegions
  ...
  public class RegionA {
    protected int a;
    protected int b;
  }
  ...
  public class RegionB {
    protected int c;
  }
  ...
}
```

Padding technique F

It appears that superclass fields are not in-lined in subclass instances. Padding technique F exploits this in order to segment regions of fields into different cache lines. The technique can be combined with technique B or E to gain more fine-grained control of the amount of padding used. The JOL output can be seen in figure 6.5

Used with inner-classes, this can be an expressive technique for segmenting fields in to separate memory regions with clear syntactical boundaries. This may result in code that is more readable than when using technique E by itself.

The Contended annotation

```
public static class PaddedClusterContended implements
  Cluster {
  @Contended("group1")
  private volatile Point mean;
  @Contended("group2")
  private double sumx, sumy;
  @Contended("group2")
  private long count;
  ...
}
```

Padding technique G

Finally, padding technique G uses an undocumented annotation `sun.misc.Contended` for this exact purpose. It is undocumented, so there

is no guarantee it will work on any particular Java runtime, or that it won't be removed in the future. However, this is true for all the padding techniques we have examined. Using the `@Contended` annotation yields a compiler warning: "warning: Contended is internal proprietary API and may be removed in a future release". It is not clear if using the annotation affects licensing.

A clear disadvantage of this technique is that we have no control over the amount of padding used. Advantages of this technique is that the runtime platform can clearly see that we intend to use padding, and may prevent the garbage collector from compacting our memory layout.

JOL's internals command does not show any results of using the `@Contended` annotation, but the output of the estimates command can be seen in figure 6.6. We can see that several large (128-132) unused memory regions have been added to the instance. The annotation takes an optional argument for the name of the field group. Fields annotated using the same argument are put in a contiguous memory segment. The `@Contended` annotation can also be used on class declarations.

```
com.kmeans.Clusters$PaddedCluster object internals:
 OFFSET  SIZE               TYPE DESCRIPTION
      0     4                    (object header)
      4     4                    (object header)
      8     4                    (object header)
     12     4                int PaddedCluster.dummy1
     16     8             double PaddedCluster.dDummy1
     24     8             double PaddedCluster.dDummy2
     32     8             double PaddedCluster.dDummy3
     40     8             double PaddedCluster.dDummy4
     48     8             double PaddedCluster.dDummy5
     56     8             double PaddedCluster.dDummy6
     64     8             double PaddedCluster.dDummy7
     72     8             double PaddedCluster.dDummy8
     80     8             double PaddedCluster.sumx
     88     8             double PaddedCluster.sumy
     96     8               long PaddedCluster.count
    104     4                int PaddedCluster.dummy2
    108     4                int PaddedCluster.dummy3
    112     4                int PaddedCluster.dummy4
    116     4                int PaddedCluster.dummy5
    120     4                int PaddedCluster.dummy6
    124     4                int PaddedCluster.dummy7
    128     4                int PaddedCluster.dummy8
    132     4                int PaddedCluster.dummy9
    136     4                int PaddedCluster.dummy10
    140     4                int PaddedCluster.dummy11
    144     4                int PaddedCluster.dummy12
    148     4                int PaddedCluster.dummy13
    152     4                int PaddedCluster.dummy14
    156     4                int PaddedCluster.dummy15
    160     4                int PaddedCluster.dummy16
    164     4                int PaddedCluster.dummy17
    168     4   com.kmeans.Point PaddedCluster.mean
    172     4                    (loss due to the next
object alignment)
Instance size: 176 bytes
Space losses: 0 bytes internal + 4 bytes external = 4
bytes total
```

Figure 6.4: JOL output for padding technique E

```
RegionA object internals:
 OFFSET  SIZE            TYPE DESCRIPTION
      0     4                 (object header)
      4     4                 (object header)
      8     4                 (object header)
     12     4             int RegionA.a
     16     4             int RegionA.b
     20     4   PaddedRegions RegionA.this$0
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0
bytes total

...

RegionB object internals:
 OFFSET  SIZE            TYPE DESCRIPTION
      0     4                 (object header)
      4     4                 (object header)
      8     4                 (object header)
     12     4             int RegionB.c
     16     4   PaddedRegions RegionB.this$0
     20     4         (loss due to the next object
alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4
bytes total
```

Figure 6.5: Output from running JOL against each of the classes in padding technique F

```
1  ***** 64-bit VM, compressed references enabled: *****
2  com.kmeans.Clusters$PaddedClusterContended object
3  internals:
4   OFFSET  SIZE    TYPE DESCRIPTION
5        0    12         (object header)
6       12   132         (alignment/padding gap)
7      144     8  double PaddedClusterContended.sumx
8      152     8  double PaddedClusterContended.sumy
9      160     8    long PaddedClusterContended.count
10     168   128         (alignment/padding gap)
11     296     4   Point PaddedClusterContended.mean
12     300   132                     (loss due to the next
13     object alignment)
14 Instance size: 432 bytes
15 Space losses: 260 bytes internal + 132 bytes external
16 = 392 bytes total
```

Figure 6.6: Section of the output from running JOL's estimates command against the `PaddedClusterContended` class in padding technique G

Chapter 7

The busyness of CPUs

False sharing is closely tied to the performance the memory hierarchy. It is true that false sharing can cause additional CPU-bound work due to retries when using optimistic concurrency, but false sharing overhead is itself purely IO bound. It is easy for software designers to overlook IO performance. Perhaps it is easier to develop an intuition for the cost of arithmetic instructions than for the cost of memory instructions. It is not just software designers however. Between 2005 and 2010 the world was obsessed with FLOPS (floating point operations per second) as a performance metric. As we will see in chapter 9, a simple read-increment-write sequence with memory barriers can take several nanoseconds — or it can take hundredths to thousands of nanoseconds if the cache line is contended. Given the high cost and importance of memory operations, especially on multicore systems, we should let memory operations be the new FLOPS, in terms comparing the performance of machine architectures.

As stated previously, "cache-friendliness" takes a new meaning in multicore software. Sequential programs benefit from data being close together in memory: CPU caches opt to prefetch memory locations that are close to already requested ones; and data is cached a full cache line at a time. Additionally, the L1 prefetchers will only prefetch data within the same 4KiB page as the instruction that cases a load [8]. When the caches cache data that is not needed by a sequential program, the overhead is minimal. Multicore programs benefit from the same cache mechanisms as sequential ones, except that sharing data between CPU cores is expensive. When the caches cause data segments to be shared between cores unnecessarily, the performance loss can be considerable.

Without using benchmarks or hardware simulations, memory related performance problems can be hard to notice and diagnose. Figure 7.1 lists a couple of different ways CPUs can waste time, and how the CPU load will look as a result. As we can see, the busy-time of a thread or CPU says very little about the usefulness or necessity of the work it performs. All of the situations in the table, except perhaps for the last one, can be caused or aggravated by false sharing of cache lines or similar problems. And yet, the resulting behaviours differ wildly. The exception is cache interference and unpredictable cache accesses. They have different causes, but are difficult to tell apart because both present as cache misses.

Cause	Effect	Thread/CPU behaviour
Lock contention	Threads wait for the lock	Idle
Optimistic compare-and-swap, repeated tries	repeatedly undoes each others' work	Busy
Cache interference (e.g. from false sharing)	Cache thrashing with little progress	Stalled
Unpredictable cache accesses	No prefetching	Stalled

Figure 7.1: Different types of CPU time-waste, and how they present.

Chapter 8

Method

To better understand the effects of memory layout, padding, and false sharing overhead on managed platforms, we perform a series of experiments in the form of benchmarks. The experiments are described in detail in chapter 9. Experiments are performed on three different platforms:

i5 A laptop with an Intel i5-4210U CPU, rated at 1.7Ghz, 2.7GHz with Intel Turbo Boost, with L1, L2, and L3 cache sizes of 32KiB, 256KiB, and 3072KiB respectively. The CPU has 2 physical cores, 4 virtual cores with hyper-threading.

i7 A Desktop with an Intel i7-4790 CPU, rated at 3.6Ghz, 4.0GHz with Intel Turbo Boost, with L1, L2, and L3 cache sizes of 32KiB, 256KiB, and 8192KiB respectively. The CPU has 4 physical cores, 8 virtual cores with hyper-threading.

Xeon A server with **two** Intel Xeon E5-2680 v3 CPUs each rated at 2.5Ghz, 3.3 with Intel Turbo Boost, with L1, L2, and L3 cache sizes of 32KiB, 256KiB, and 30720KiB respectively, for a total of 60MiB of cache. Each CPU has 12 physical cores, 24 virtual cores with hyper-threading, for at total of 48 virtual cores.

The i5 and i7 platforms run Arch Linux, and experiments are performed on the OpenJDK Runtime Environment, version 1.8.0_172. The Xeon platform runs Windows 10, and experiments are run on the Oracle Java(TM) SE Runtime Environment, version 1.8.0_144.

All platforms run 64-bit versions of Java.

Using OpenJDK on two platforms and Oracle's runtime on the third is not quite rigid enough to show us the differences there may be between the two runtimes, but it lets us see that our observations and techniques a valid for both of them.

Benchmarks are run using the `mark8` method from Sestoft's microbenchmarking framework [22], adapted to include a warm-up phase.

Using benchmarks on a managed platform like java limits our ability to interpret our results: We cannot examine the nature and cause of individual memory operations. Instead, we benchmark multiple variations of the same programs, and attempt to interpret the results in terms of the variations. This involves

a certain amount of guesswork, and we cannot be sure that the differences between two implementations do not effect unintended changes to the runtime behaviour. Similarly, we cannot know the actual memory layout at runtime, due to the possibility of heap-fragmentation etc. We can only guess the layout effects of our optimizations, and see if our results agree with our assumptions about the memory layout.

Our results are also subject to noise from other programs. While attempts have been made to disable periodic operating-system processes as well as application-software on all platforms, it is almost certain that our experiments are not allotted all hardware resources when they run. In chapter 9 we measure the idle CPU load of each platform, to convince ourselves that the noise levels are insignificant.

Chapter 9

Experiments

In this chapter we look at multiple examples of multicore programs, focusing on how they are affected by false sharing of cache lines. We start with a few slightly contrived programs in section 9.2, to show the extremes of false sharing overhead. We then move on to more practical examples such as histogram building, sorting, and k-means clustering in section 9.3

Unless otherwise noted, experiments are run with 4, 8, and 48 threads on the i5, i7, and Xeon platforms, respectively. These numbers were chosen based on the experiments in the following section.

9.1 Platform experiments

Before we get into our multicore performance experiments, it is worth verifying some assumptions about the hardware platforms we perform them on. In the following sections, we first assess noise in our experiments by examining the CPU load when not running experiments. We then gauge our platforms' effective or perceived degree of parallelism, by measuring how well simple tasks scale with the number of threads used. We finally measure the access times of the different levels of the cache hierarchy.

9.1.1 CPU idle load

Our experiments do not run on bare metal: The underlying operating system and other user-applications may put load on the CPU, and skew our results. To assess the amplitude of this noise, we measure the CPU load on the platforms when not running experiments. The CPU load information is gathered from the Linux top utility on the i5 and i7 platforms, and from the Windows typeperf command on the Xeon platform.

Values from both tools are sampled over 1-second intervals. The top utility gives integer values, while typeperf gives decimal values.

The average values for the durations of the experiments are: 0.20% for the i5 platform, 0.02% for the i7 platform, and 0.09% for the Xeon platform.

Figure 9.1 shows the CPU load of all three platforms when no experiments are running.

The values are given as percentages of a load that would keep all cores busy 100% of the time. Therefore, the Xeon platform is actually the busiest, despite not having the highest average. Expressing the load averages as percentages of a load that would keep a single, virtual core busy, the loads are 0.79%, 0.16%, and 4.14% for the i5, i7, and Xeon platform respectively.

For the performance effects we wish to examine in the remainder of this report, this level of noise seems to be acceptable.

9.1.2 Degrees of parallelism

It is not entirely clear how much work our platforms can do in parallel. We know how many physical cores they each have, and we know how many and virtual cores they each have with hyper-threading. But hyper-threading is not the same as having additional physical cores: Virtual cores share hardware such as caches, do not fully execute two threads at once. Rather, hyper-threading allows two threads to interleave on a single physical core, so that when one thread is waiting for IO, the other may perform CPU bound work, and vice-versa.

To find the effective or perceived degree of parallelism our platforms support, we run four experiments with different load types: IO bound, CPU bound, and mixed load.

Each experiment runs a fixed number of iterations for each of the platforms. Work load is divided between evenly threads, so more threads means fewer iterations per thread.

The experiments are:

mem-reads A memory bound experiment. Each thread reads elements from an array and keeping a running sum on the stack. The array is traversed in random order, and contains random integers. The array size length is chosen as twice the L3 cache size for the given platform. The code for this experiment is shown in code snippet 9.1. The loop runs for 40E6, 32E6, and 64E6 iterations on the i5, i7, and Xeon platforms respectively.

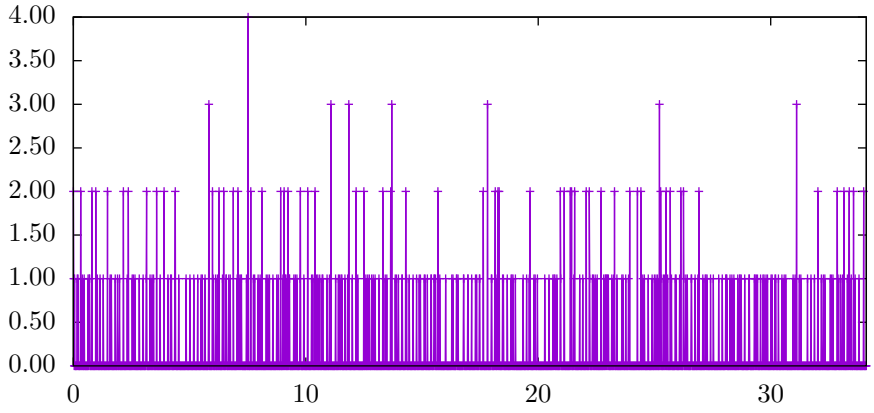
mem-writes The exact same as mem-reads, but the sum is written back to the array in each iteration. The loop runs for 30E6, 32E6, and 64E6 iterations on the i5, i7, and Xeon platforms respectively.

CPU The CPU bound experiment, shown in code snippet 9.2. Each thread performs a sequence of floating-point divisions. The divisor is a value greater than 1, chosen such that `d` never becomes less than or equal to 1. The loop runs for 40E6, 108E6, and 750E6 iterations on the i5, i7, and Xeon platforms respectively.

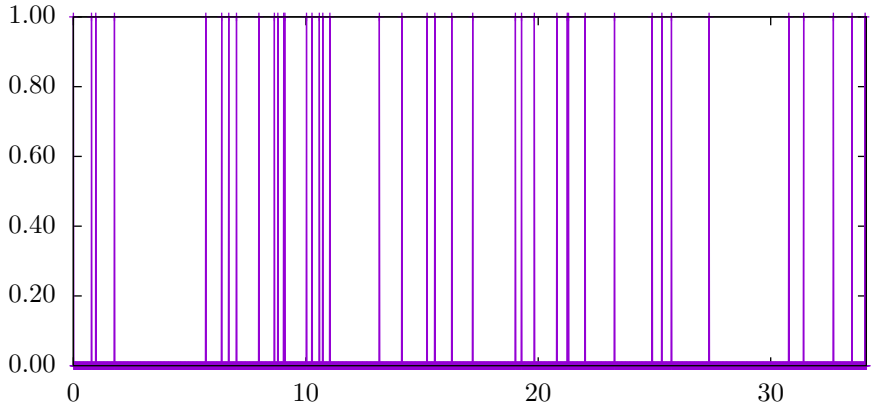
mixed A mixed-load experiment. This experiment uses half of the threads to run the mem-reads experiment, and the other half to run the CPU experiment. Since the experiments divides work between the threads they are allotted, the mixed experiment performs twice as much work per thread as the other experiments.

Figures 9.2, 9.3, and 9.4, show the wall-clock times of the four experiments on the i5, i7, and Xeon platforms respectively.

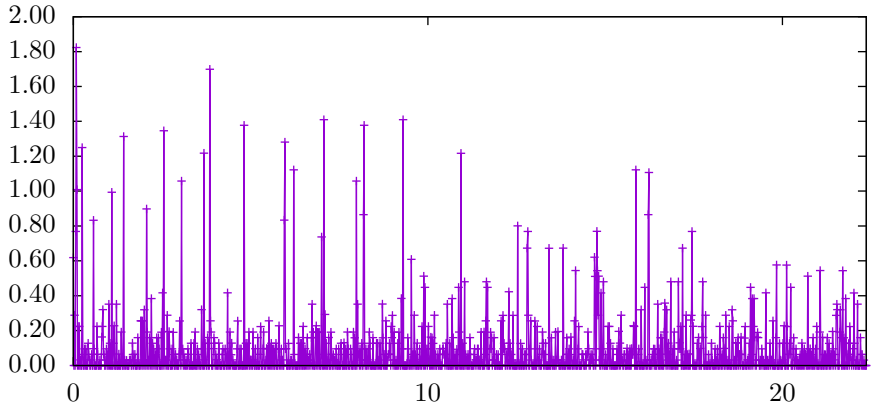
The mixed experiment parallelises the best (remember that it divides its threads between the mem-reads and CPU experiments, which means that the



(a) i5



(b) i7



(c) Xeon

Figure 9.1: CPU baselines for benchmarks. The x-axes show CPU load in percent, as reported by the operating system (100% load corresponds to all cores being busy 100% of the time). The y-axes show time in minutes. Please note that both axes differ between the plots.

mixed experiment for 2 threads performs as much work as the mem-reads and CPU experiments do together for 1 thread, and so forth). This agrees with our intuition of hyper-threading works.

Across the different types of loads, we see that 4, 8, and 48 threads are good thread counts for the i5, i7, and Xeon platforms respectively.

```
let taskCount parallel tasks do {
  //start indexes are evenly spaced for the threads
  final int startIndex = ...;
    double sum = 0.0;
    int k = startIndex;
    for (int j=0; j < workPerThread; j++){
      if(k<arraySize - 1){
        ++k;
      } else {
        k=0;
      }
      sum += array[indices[k]];
    }
}
```

Code snippet 9.1: Simplified code for the memory-bound parallelism experiment. The [indices] array contains all the indices of the `array` array in random order.

```
let taskCount parallel tasks do {
  double d = Double.MAX_VALUE;
  for (long j = 0; j < workPerThread; ++j) {
    d /= divisor;
  }
}
```

Code snippet 9.2: Simplified code for the CPU-bound parallelism experiment.

9.1.3 Memory hierarchy access-times

Here we replicate the experiment used to measure read access times in [3]. Code snippet 9.3 shows the code for the core measurement loop, figure 9.5 shows the results. The experiment performs a fixed number of memory read operations ($2^{25} = 33.54432E6$) from an array. The access pattern is determined by the contents of a pre-built array. The pattern is cyclic, so the number of read operations and the working-set size are independent of each other, and randomized to hinder cache prefetching. This allows us to measure the read times of the different levels of the memory hierarchy, by choosing working set sizes that are too large to fit in the smaller levels. If we e.g. wish to measure the access times for L2 cache, we use a working set larger than the 32KiB that

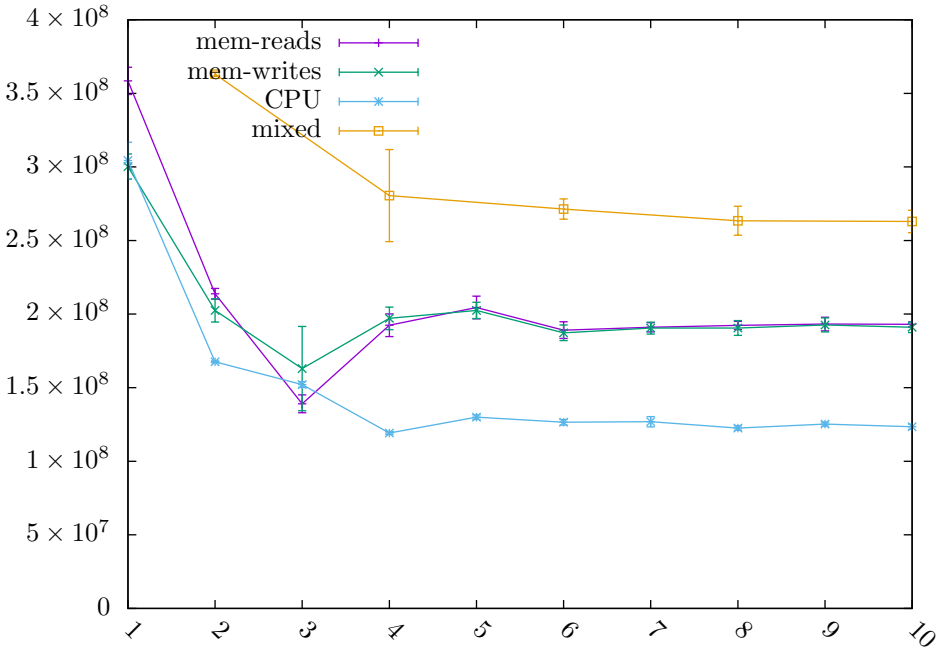


Figure 9.2: Parallelism experiment on the i5 platform. The plot shows wall-clock execution time in ns., as a function of thread count.

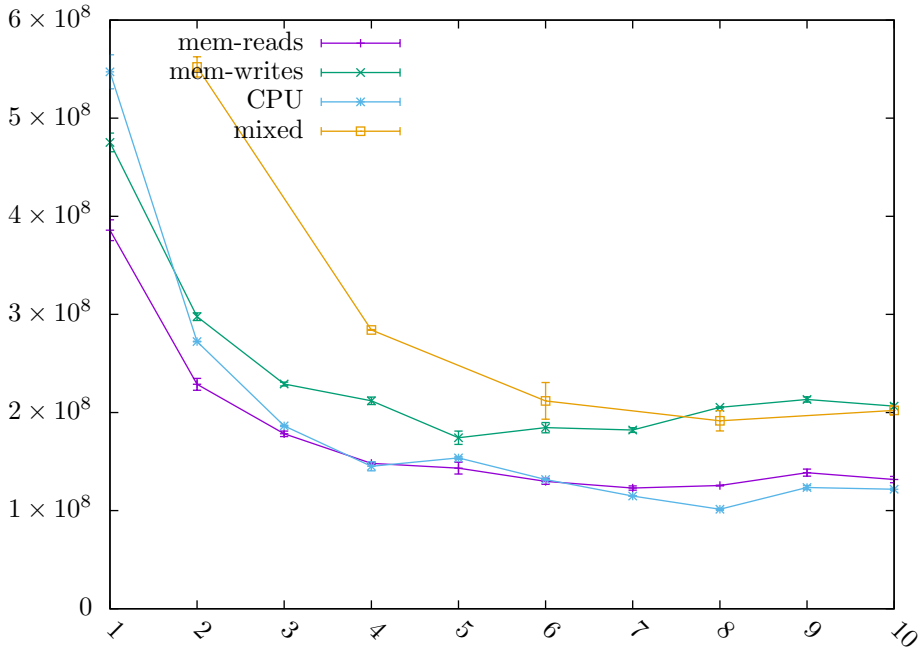


Figure 9.3: Parallelism experiment on the i7 platform. The plot shows wall-clock execution time in ns., as a function of thread count.

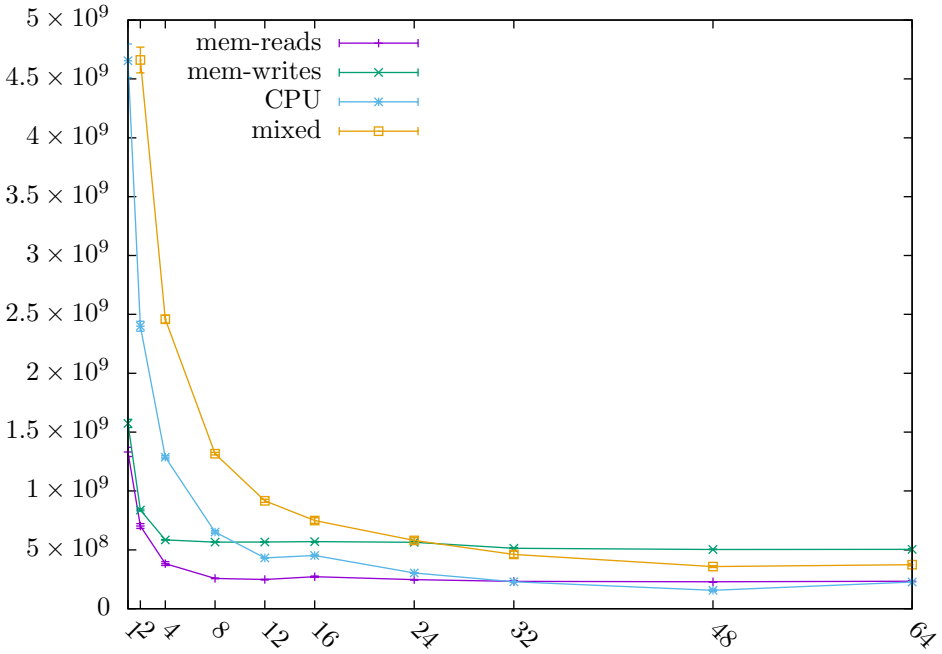


Figure 9.4: Parallelism experiment on the Xeon platform. The plot shows wall-clock execution time in ns., as a function of thread count.

will fit in L1, but no larger than the 256KiB that will fit in L2. The technique used to fill the array, thereby deciding the access pattern, is rather intricate, so rather than restating it here, I refer interested parties to [3].

Results are included for an additional padded variant that uses padding technique C. This is intended as a sanity-check to ensure that each read array element is in its own cache line. The idea is to avoid values being cached because they share a cache line with previously read values. As the results show, this is unnecessary: The technique from [3] already achieves this, as the randomized access pattern causes the previously read cache lines to be evicted. This is evident by the fact that the padded versions are approximately twice as slow as the unpadded ones, corresponding to the additional memory read from the indirection array used for padding.

```
private static double jumps(int[] arr) {
  int k = 0;
  for(int j = 0; j < 1 << 25; ++j){
    k = arr[k];
  }
  return k;
}
```

Code snippet 9.3: Code for measuring memory access times.

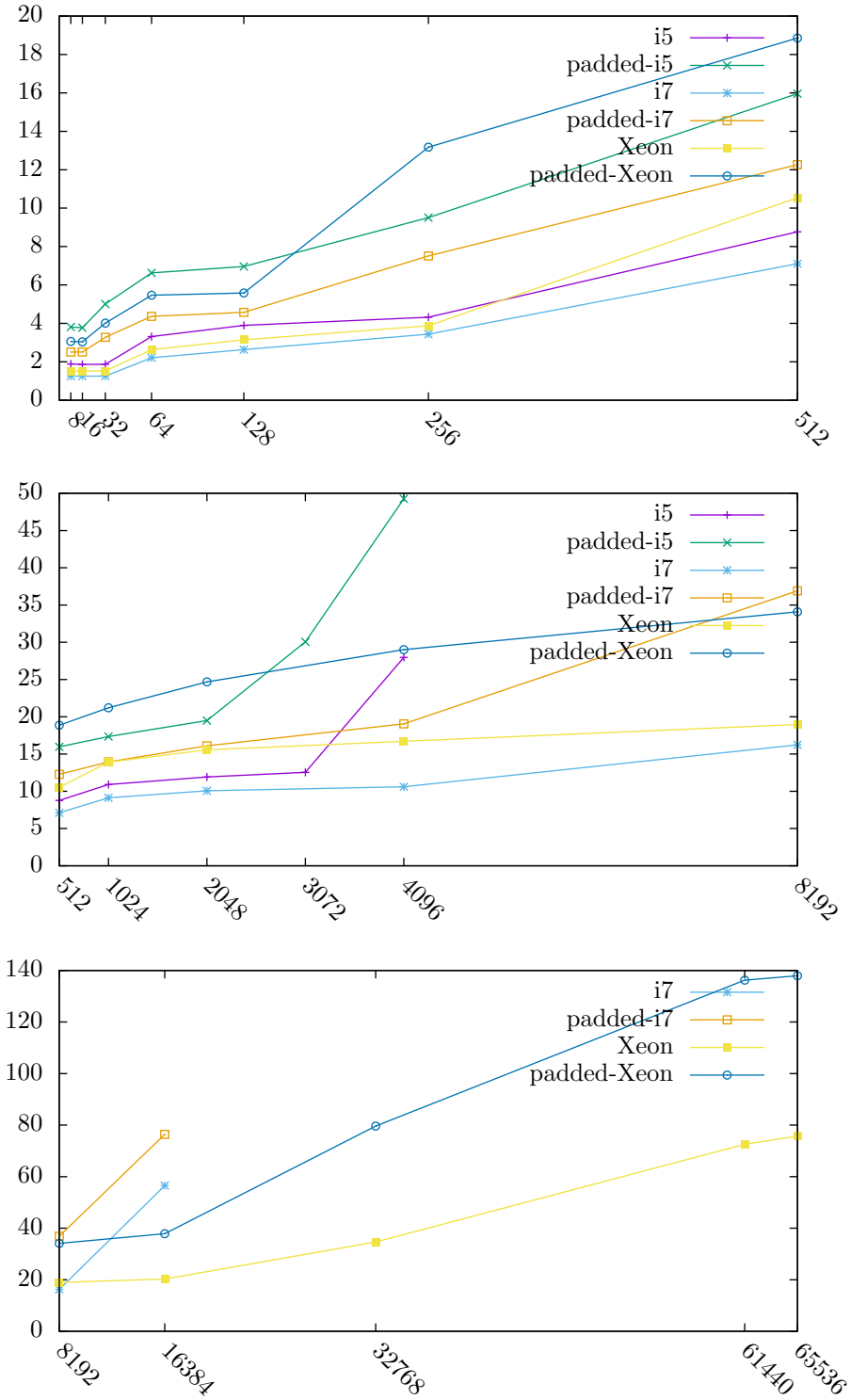


Figure 9.5: Random cyclic reads, approximating memory-access times on all 3 platforms. The y-axes show the average time per read operation, measured over 2^{25} reads. The x-axes show the working-set size in KiB.

9.2 Multicore cache performance

To see the impact of false sharing, it is illustrative to look at a few contrived example programs. In this section we look at a handful of variations of programs that perform simple operations in a multicore setting.

9.2.1 Uncontended writes

The first example we examine illustrates the impact of false sharing by running simple, uncontended, integer-field increment operations in parallel threads. To see the impact of false sharing, we observe the time it takes to perform an increment as a function of the distance between the fields in memory.

```
let taskCount parallel tasks do {
  Counter cc = ...;
  for (long i = 0; i < increments_per_thread; i++) {
    cc.value++;
  }
}
```

Code snippet 9.4: Simplified code for the local-field version of the uncontended-writes experiment.

Each thread performs integer-increment operations. The fields are uncontended; each thread has its own integer-field and performs no read or write operations to fields used by the other threads. We run two variants of the experiment, one where the `Counter` instance with the integer field is kept in a thread-local field, and one where it is read from a shared array in each iteration. Since each thread operates on its own `Counter` instance, there is no data sharing between CPU cores, and hence no need for synchronization. Nonetheless, we perform the experiment in variants with and without `volatile` integer declarations, to see the performance impact in both cases.

Since the fields are uncontended, we will assume the difference in performance to be a result of unnecessary coherence operations due to false sharing.

In the experiments with `volatile` fields, each thread performs 6E6 increments. In the experiments without `volatile`, each thread performs 66E6 increments. Figure 9.6 and 9.7 show the wall-clock execution time of the experiments, as a function of padding. Padding technique B was used for these experiments. The values chosen for the amounts of padding are a little odd. This is due to a bug in the padding code that was discovered late in the project.

The results show significant overhead from false sharing. With volatile fields, padding reduces execution time by 78-94%. With non-volatile fields, padding reduces execution time by 86-94%, but only for the array versions. For the local versions with non-volatile fields, padding reduces execution time by only 12-19%. The threads never write to the array, only to the integer fields of the objects pointed to by the array elements, so no explanation for this behaviour presents itself.

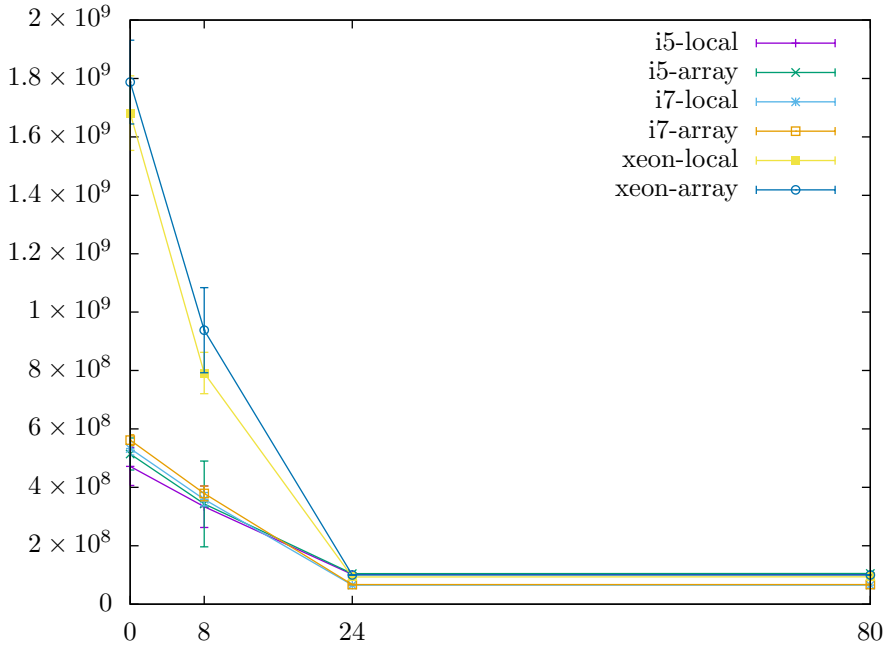


Figure 9.6: Uncontended increments on volatile integer fields. The plot shows wall-clock execution time, as a function of the number of bytes used as padding between the integers.

9.2.2 Contended writes

In the previous section we examined the performance of uncontended memory operations to see the existence and cost of false sharing. In this section, we shall see that observing the performance of contended memory operations - where coherence overhead is strictly necessary - can be just as illustrative.

```
int sharedInt = 0;
let taskCount parallel tasks do {
  for (long j = 0; j < increments_per_thread; ++j) {
    sharedInt++;
  }
}
```

Code snippet 9.5: Simplified code for the local-field version of the contended-writes experiment.

The code, seen in snippet 9.5, is similar to that in snippet 9.4, but in this version all threads operate on a single, shared integer field. As in the previous section, we run two versions of the experiment: One where the field is `volatile`, and one where it is not. As this experiment uses a shared field, there is no need to store it in an array. However, benchmarks using a single-element array are

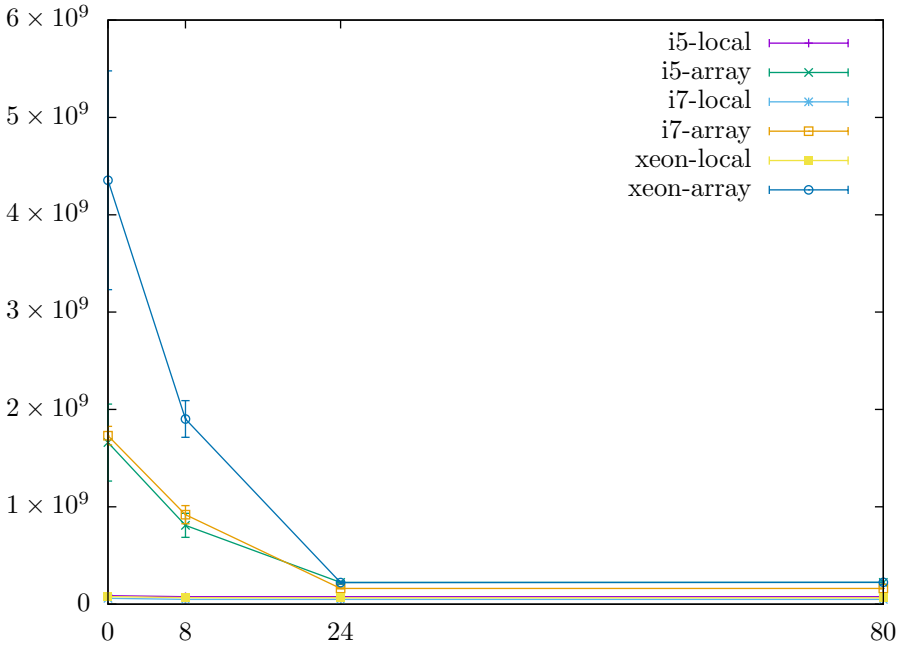


Figure 9.7: Uncontended increments on non-volatile integer fields. The plot shows wall-clock execution time, as a function of the number of bytes used as padding between the integers.

included, to show that the behaviour is not significantly affected by this change.

An additional experiment is included here, running an additional thread that only reads the integer field. This experiment is the exact same as described in code snippet 9.5, except that when it runs with e.g. four threads, there is an additional fifth thread running that only performs reads. This allows us to see that, perhaps contrary to intuition, read operations in a multicore setting also incur a coherency overhead.

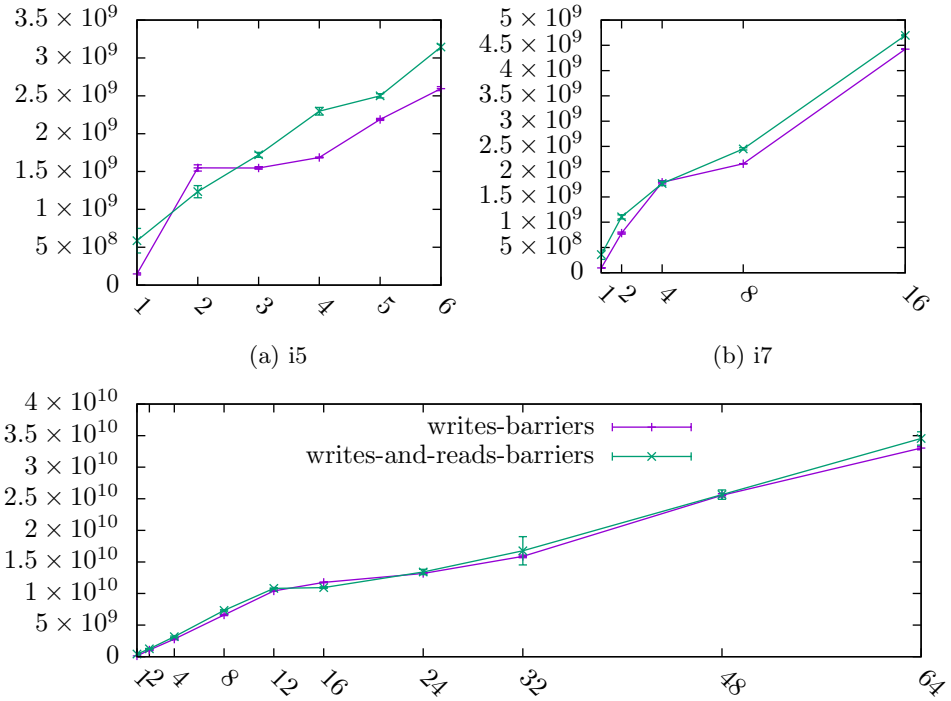
We need the reading thread to run for the full duration of the experiment. To that end, the thread does not perform a predetermined number of operations, but is started before the others, and terminated only when the others are finished. This incurs an overhead, as the thread needs to be stopped before the benchmark finishes! However, an additional experiment shows that the time it takes to stop a thread is far too small to significantly skew our results:

The experiments with a volatile field runs 20E6 iterations of the loop; the experiments with a non-volatile field runs 200E6 iterations of the loop.

The results, shown in figures 9.9 and 9.10 show that memory reads can have a notable performance impact. Both with and without the `volatile` field declaration, we see that adding the additional reading thread generally increases execution time. In this respect, the plots can only really be trusted up to about 3, 8 and 48 threads on the i5, i7, and Xeon platforms, as this about the degree of parallelism we can expect, as shown in the parallelism experiments previously. With volatile fields, running one writing and one reading thread takes 3.27-3.97 times as long as running a single writing thread. Running two writing threads

Platform	Time (ns)	SD (%)
i5	833.4	0.33
i7	1940.5	3.00
Xeon	1256.7	1.08

Figure 9.8: Wall-clock time for for stopping a thread. Time is the total wall-clock execution time, the standard deviation (SD) is given in percent of the execution time.

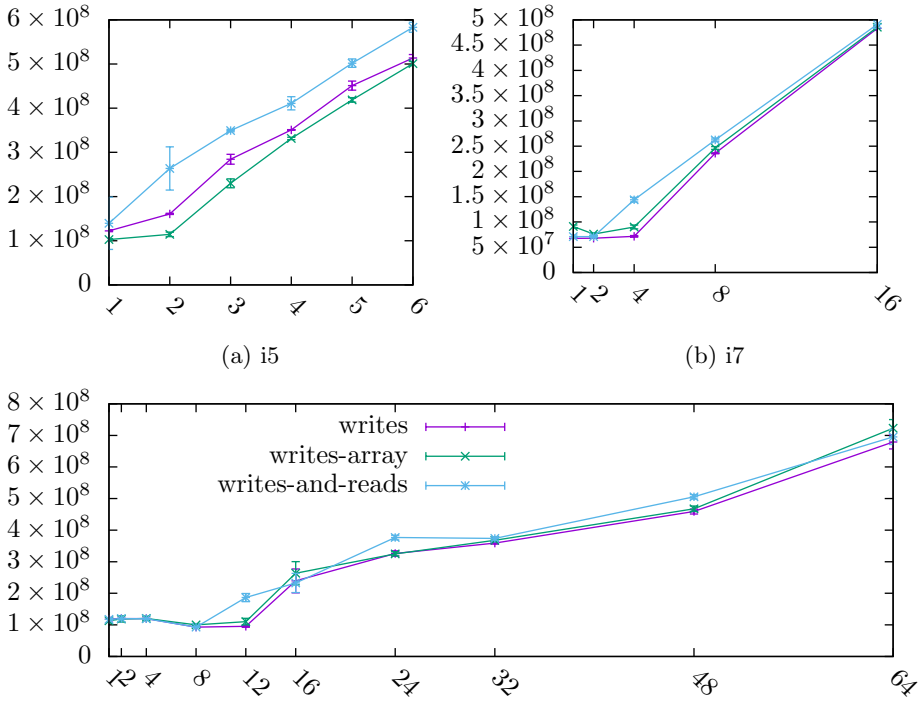


(a) i5

(b) i7

(c) Xeon

Figure 9.9: Contended increments on **volatile** integers. The plots show wall-clock execution time in nanoseconds, as a function of the thread count. Please note that both axes vary across the plots.



(a) i5

(b) i7

(c) Xeon

Figure 9.10: Contended increments on **non-volatile** integers. The plots show wall-clock execution time in nanoseconds, as a function of the number of threads. Please note that both axes vary across the plots.

is, of course, even slower, but it is a significant slowdown from something we might intuitively expect to have no additional cost. It seems the extra cache invalidations needed to maintain cache coherence are the culprits. Without the `volatile` field declaration, running one writing and one reading thread takes 14% longer on the i5 platform, and 5% longer on the i7 platform. On the Xeon platform, the additional reading thread makes the experiment run 0.6% faster. This is likely just noise. It is not clear why reading from an array is faster than reading from a field on the i5 platform.

Just as the extra cost of the writing thread is interesting, it it interesting to see the cost of additional writing threads. On the i5, i7, and Xeon platforms, running with two writing threads is 10.48, 8.19, and 8.06 times slower than running with a single writing thread. This loosely illustrates the cost of passing cache lines back and forth between two cores: 70.00, 34.39, and 46.50 nanoseconds per increment on average.

9.3 Practical applications

While incrementing counters can be useful, it is thankfully not the only thing we build software for. In this section we examine false sharing in 4 examples of more practical parallel programs.

First we will look at several programs that build histograms. These example programs will motivate our approach from coarse- to fine-grained synchronization with padding. One implementation also serves as an example of false sharing in lock-free application based on optimistic concurrency with compare-and-swap. We then use Quicksort as an example of the divide and conquer paradigm, where the division of subproblems limit the need for synchronization. As examples of locking applications, we examine an implementation of k-means clustering, as suggested in the January 2017 exam in Practical Concurrent and Parallel Programming (PCPP) at the IT University of Copenhagen [23, 24], as well as two implementations of a concurrent hashmap data structure used in the same course. Both of these applications are studied in [3], and can be said to have instigated this project.

9.3.1 Histogram builder

Let us consider the problem of building a histogram: Given a sequence of integers as input, we wish to build a data structure that maps numbers to their frequencies in the input sequence.

A simple parallel algorithm for building a histogram presents itself: Divide the sequence into a number of equal-sized sections, one for each thread. For each element a in its section, have each thread atomically increment a global counter (or bucket) representing the frequency of a.

Except for the first, all the implementations we consider follow this basic formulation; they differ only in how they ensure that the increment operations are atomic. We need to ensure atomicity because the numbers in the input sequence may appear multiple times, and in different segments, making it possible for more than a single thread to increment the same frequency counter at the same time.

The histogram benchmarks are performed with the following parameters: An input sequence which consists of 4 million randomly chosen integers $a \in [0, 31]$, and is divided evenly between threads. The more threads, the less work per thread. We use 32 buckets: One for each possible number in the input. We will refer to the number of buckets as the "width" of the histogram.

Communication-free

As a baseline for the multicore performance of the histogram problem, we examine an implementation with minimal communication overhead. The strategy employed can be thought of as a kind of avoidance or confinement: We arrange it so that part of the problem can be solved in parallel, *without ongoing communication between cores*. After the parallel step, a single thread consolidates the thread-local results produced by each thread. The only necessary communication or synchronization is making the thread-local results visible to the thread that executes the consolidation step, *after* the parallel step is completed. Code snippet 9.6 outlines the implementation of this approach.

```
List<Counter[]> perThreadCounts = ...;
// Perform thread-local counts in parallel
let taskCount parallel tasks do {
  // Thread-local counter array. A reference to the
  // same array is stored in perThreadCounts
  Counter[] counters = ...;
  final int from = ..., to = ...;
  for(int j = from; j < to; ++j) {
    counters[inputSequence[j]].value++;
  }
}
// Consolidate per-thread counts sequentially
Counter[] globalCounts = new Counter[WIDTH];
for(Counter[] localCounts : perThreadCounts) {
  for(int i = 0; i < WIDTH; ++i){
    globalCounts[i].value += localCounts[i].value;
  }
}
```

Code snippet 9.6: Simplified code for the communication-free version of the histogram builder.

As we will see, this is by far the fastest of our solutions to the histogram. It also scales relatively well with the number of cores.

Coarse-grained locking

When it comes to locking implementations, the simplest solution is for the threads to take a single, shared lock anytime they increment a frequency counter. This solution is slow. At any time, only one thread can be incrementing a counter.

Platform	Time (ms)	SD (%)
i5	18.50	2.34
i7	7.67	1.71
xeon	5.00	0.71

Figure 9.11: Execution times for the communication-free histogram builder. Time is the total wall-clock execution time, the standard deviation (SD) is given in percent of the execution time.

```
Object lock = new Object();
let taskCount parallel tasks do {
  final int from = ..., to = ...;
  for(int j = from; j < to; ++j) {
    synchronized(lock){
      counters[inputSequence[j]].value++;
    }
  }
}
```

Code snippet 9.7: Simplified code for the threads in the coarse-grained locking version of the histogram builder.

The table in figure 9.12 shows the execution times of this solution to the histogram problem. At we would expect, it scales poorly with the number of cores.

Platform	Time (ms)	SD (%)
i5	242.62	12.89
i7	340.86	8.06
xeon	492.04	11.67

Figure 9.12: Execution times for the histogram builder using a single global lock. Time is per input-element per thread, the standard deviation (SD) is given in percentage of the execution time.

We will not measure it, but there is a possibility for false sharing in this implementation. Consider the following sequence of operations, where we assume counter0 and counter1 are in the same cache line:

1. CPU0 updates counter 0. The cache line is left in CPU0's cache.
2. CPU1 updates counter 1. The cache line is invalidated in CPU0's cache.
3. CPU0 updates counter 0. The counter is invalid in CPU0's cache, and must be read from CPU1's cache instead.

The cache miss in step 3 is unnecessary. It only occurs because the two counters share a cache line, and because the other counter was updated in step 2

by a different CPU core. The impact of false sharing here is likely overshadowed by the poor choice of locking scheme.

There is another kind of unnecessary/false communication between CPU cores here, that has nothing to do with sharing of cache lines: When a thread takes the lock, updates from previous threads are guaranteed to be made visible to it. Except for updates to the counter the thread is *trying* to increment, this is unnecessary.

Fine-grained locking

A better solution is to have different counters guarded by different locks. This way, threads can perform their work in parallel, except when different threads work on counters guarded by the same lock. For practical applications the ideal degree of granularity should be determined by experiment. In this experiment I use 32 locks: One lock for each bucket.

```
Object[] locks = ...;
let taskCount parallel tasks do {
  final int from = ..., to = ...;
  for(int j = from; j < to; ++j) {
    int a = inputSequence[j];
    synchronized(locks[a]){
      counters[a].value++;
    }
  }
}
```

Code snippet 9.8: Simplified code for the threads in the fine-grained locking version of the histogram builder.

The improved level of parallelism increases the possibility for false sharing: In the previous version only one thread could perform writes at a time. In this version, threads holding different locks can in theory spend arbitrary amounts of time invalidating each other's cache lines without doing actually incrementing the counterss.

There are two clear candidates for false sharing: The locks and the counters. Neither are stored directly in the arrays: Locks must be object types, and are hence stored as references, and the integers used for the counters are stored in placeholder objects so we can declare them as `volatile`. False sharing of the references stored in the arrays should therefore be irrelevant, but if the objects are densely allocated, e.g. in a tight for-loop, they can still be placed back-to-back in memory.

We measure the impact of false sharing by benchmarking versions with different amounts of padding between the locks and counters. We use padding technique D for this experiment.

Figures 9.14, 9.15, and 9.16 show the wall-clock execution times using different amounts of padding for the i5, i7, and Xeon platforms respectively. Each of the smaller plot shows the execution time of the histogram builder, as a function

Platform	Time wo. padding (ms)	Best time (ms)	Improvement
i5	113.0	100.88	10.7%
i7	71.88	59.9	16.9%
Xeon	420.8	118.62	71.8%

Figure 9.13: Best times vs. times without padding for the fine-grained histogram builder. Times are wall-clock execution times. Improvement is given in percent of the unpadded execution time: An improvement of 71.8% indicates that at least 71.8% of the execution time is spend on unnecessary coherence communication in the unpadded version.

of the amount of padding between the counters. The third axis – the amount of padding between the locks – is unrolled into the different plots.

The table in figure 9.13 compares the unpadded times with the best times for each platform.

The effect is smallest on the i5 platform: Using 128 bytes padding between counters, and no padding between locks, yields a 10.7% improvement on not using padding. Padding the locks does not seem to benefit performance, and even makes it worse in some cases.

On the i7 platform, padding locks and counters both benefits performance. Padding 128 bytes between counters and 112 bytes between locks yields a 16.9% percent improvement on not using padding.

The most significant effect is seen on the Xeon platform. Like on the i7 platform, using padding is beneficial for both data structures, but here the benefit from padding the locks is much more pronounced. Using 128 bytes of padding for the counters and 112 bytes for the locks yields a 71.8% improvement on not using padding, indicating that most of the time is spent on cache coherence overhead in the unpadded version.

There is a noteworthy caveat: The experiment runs with 48 threads on the Xeon platform, but there are only 32 locks. This cannot not be solved simply by increasing the number of locks, as there are still only 32 distinct values in the input, and therefore only 32 counters. The scarcity of locks likely limits throughput, but the false sharing impact should be the same without this limitation, which is what we are concerned with.

It is not surprising that the platform with the most cores suffers the worst. The more threads running in parallel, the bigger the likelihood of invalidating a cache line that is relevant to another core. Furthermore, the invalidation messages must be sent to all the other cores. Processors must wait for acknowledgement messages from all the other cores, so there is a lot more communication occurring due the coherence protocol. The larger cache hierarchy likely also means larger physical distances, which in turns makes communication slower.

While our observations seem to agree with our understanding of false sharing, two new mysteries present themselves: It is not clear why the execution time keeps improving past 64-bytes padding, and it is not clear why padding the locks has such a modest impact.

We would expect the performance to stop improving abruptly around 64 bytes, as that is the cache line size used by our hardware architectures. One possible explanation for the continued improvements is the cache prefetch mechanism. It is possible that the prefetch mechanism causes more than a single

cache line to be fetched. This is the explanation given for why the `@Contended` annotation causes 128 bytes of padding to inserted on OpenJDK, under the assumption that cache lines are half that size[25]. This explanation is in agreement with the Intel optimization manual[8], which states that the spatial prefetcher "strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk"[1]. While this prefetching does not incur a false sharing overhead, it will leave an additional cache line in the CPU's cache. Another CPU might then later write to that cache line, incurring unnecessary coherence overhead. The additional padding does not prevent the prefetcher from loading the irrelevant cache line, but it does prevent other threads from ever writing to it. We can think of this problem as false sharing of the chunks of the L2 cache's spatial prefetcher.

An explanation for the other mysterious artifact, the modest impact of padding the locks, is more elusive. It seems to make sense that using a small number of threads compared to the number of locks limits the impact of false sharing, as it lessens the likelihood of threads working on the same cache lines. This intuition fails to explain why the impact of padding the counters is larger: The memory layout of the locks should be exactly the same as that of the counters, and there are equally many of them.

Compare-and-swap

Following the same strategy as the fine-grained locking implementation , we can use optimistic concurrency to implement a lock-free version of the histogram.

Optimistic compare-and-swap works best when the operation we perform is cheap, and the resource we perform it on has low contention. That way, retries will happen rarely, and be cheap when they do.

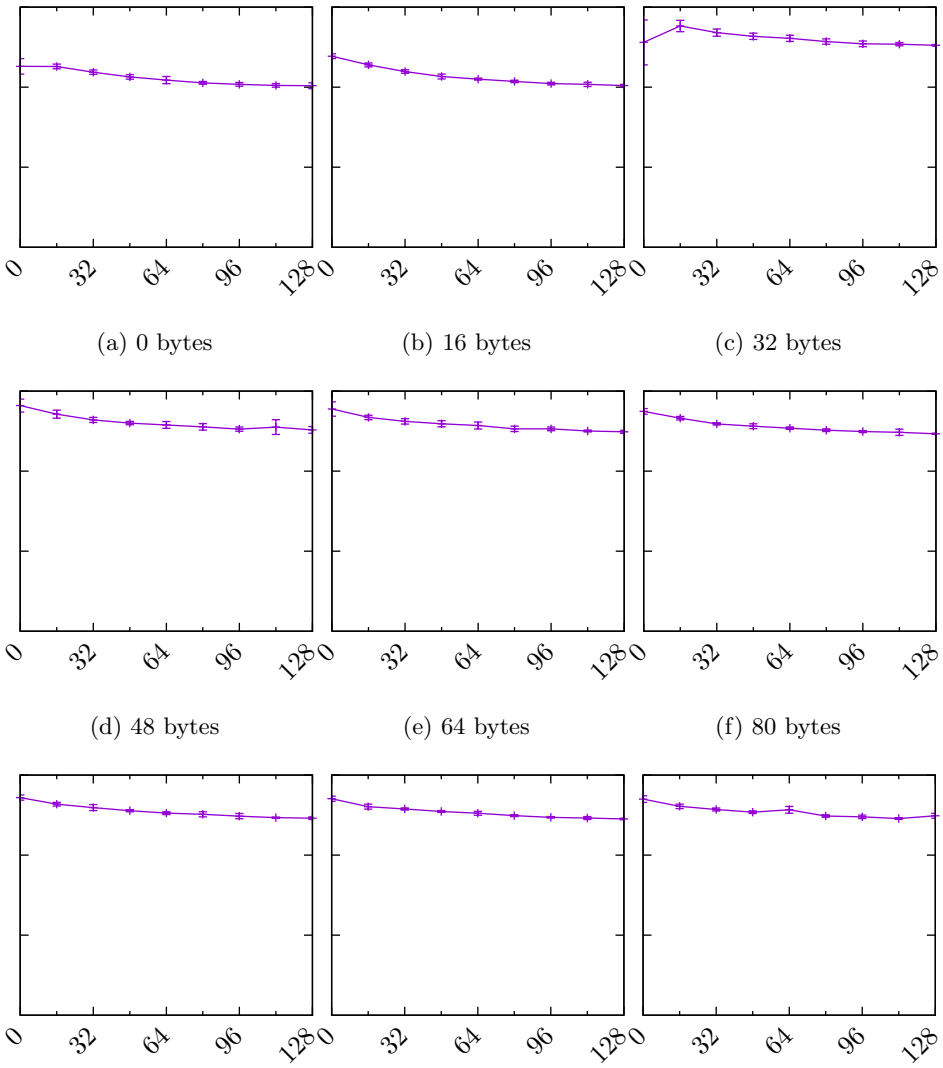
The histogram problem lends itself well to a compare-and-swap based solution: All writes to shared data are the results of simple increment operations, which are fast. Resource contention depends on the ratio of counters to threads, and the order of the input.

```
private AtomicIntegerArray counters = ...;
let taskCount parallel tasks do {
  final int from = ..., to = ...;
  for(int j = from; j < to; ++j) {
      counters.getAndIncrement(inputSequence[j]);
  }
}
```

Code snippet 9.9: Simplified code for the threads in the compare-and-swap version of the histogram builder.

Code snippet 9.9 outlines the compare-and-swap implementation of the histogram builder. It uses Java's `AtomicIntegerArray` class to hold counters.

[1]It is not entirely clear from the manual which microarchitectures this applies to. It is stated for the Sandy Bridge microarchitecture, and seems to apply to those that succeed it as well.

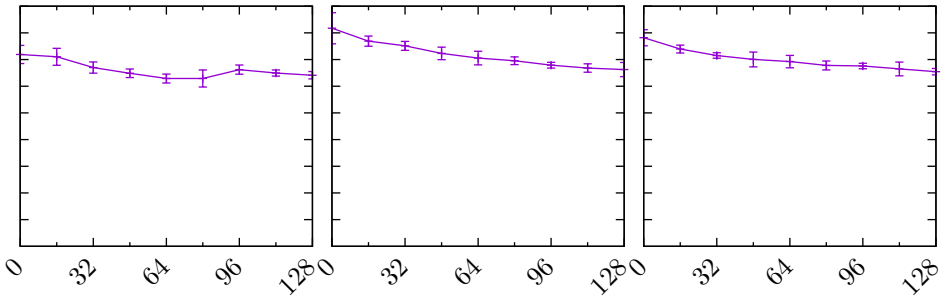


(a) 0 bytes (b) 16 bytes (c) 32 bytes

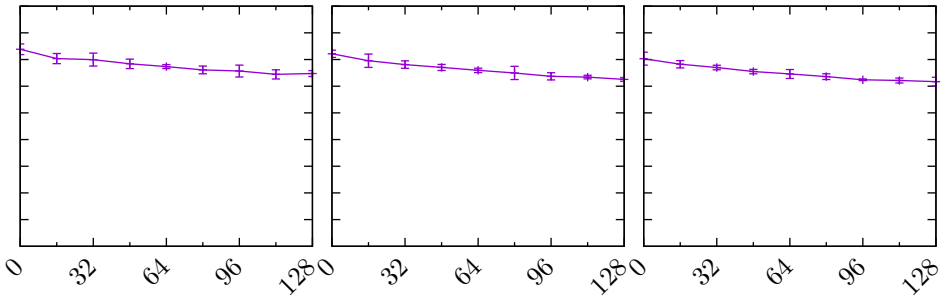
(d) 48 bytes (e) 64 bytes (f) 80 bytes

(g) 96 bytes (h) 112 bytes (i) 128 bytes

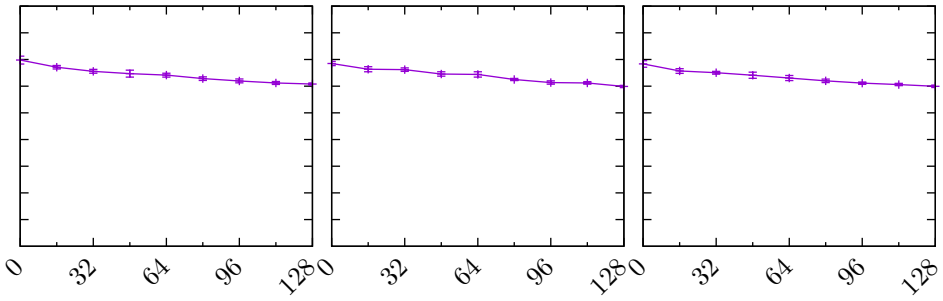
Figure 9.14: The fine-grained histogram problem on the i5 platform. Here the impact of padding the locks is negative, while padding the buckets reduces execution time by 10.7%. The best time is 100.88 milliseconds, achieved by padding the buckets with 128 bytes, and not padding the locks. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the wall-clock execution time of the histogram problem, as a function of the amount of padding between the histogram buckets in bytes. Each y-tick is 50 milliseconds. The axes starts at 0.



(a) 0 bytes (b) 16 bytes (c) 32 bytes

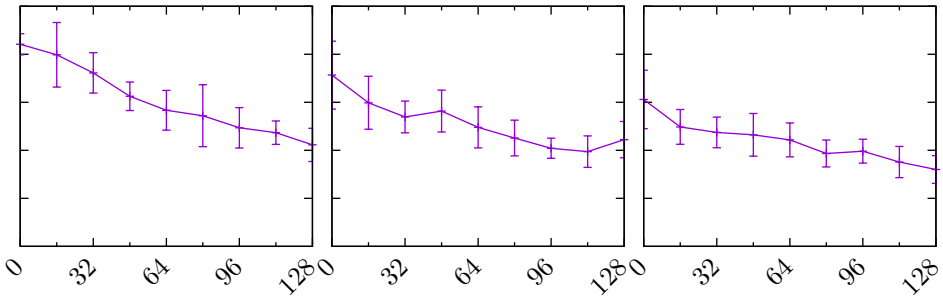


(d) 48 bytes (e) 64 bytes (f) 80 bytes

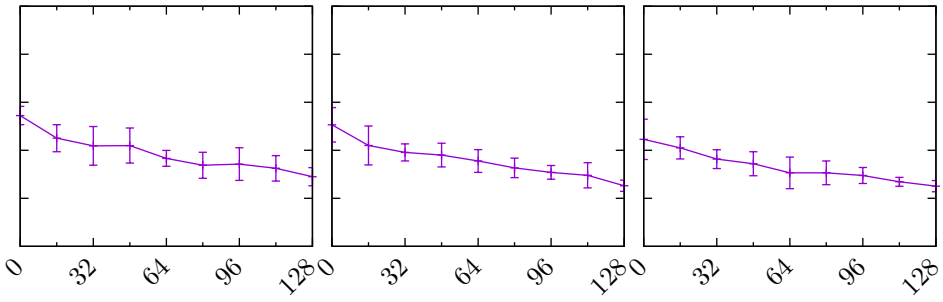


(g) 96 bytes (h) 112 bytes (i) 128 bytes

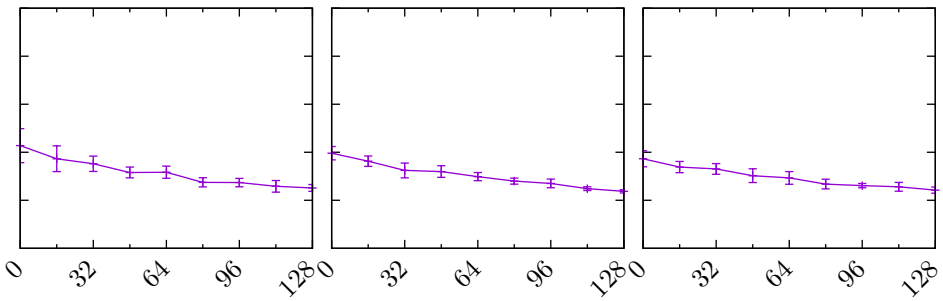
Figure 9.15: The fine-grained histogram problem on the i7 platform. Here the overall trend is that padding improves performance with respect to both buckets and locks. The best time is 59.9 milliseconds, achieved by padding the locks with 112 bytes, and the buckets with 128 bytes. Padding both buckets and locks hence reduces execution time by 16.9%. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the wall-clock execution time of the histogram problem, as a function of the amount of padding between the histogram buckets in bytes. Each y-tick is 10 milliseconds. The axes start at 0.



(a) 0 bytes (b) 16 bytes (c) 32 bytes



(d) 48 bytes (e) 64 bytes (f) 80 bytes



(g) 96 bytes (h) 112 bytes (i) 128 bytes

Figure 9.16: The fine-grained histogram problem on the Xeon platform. Here padding has a large effect, both for the buckets and the locks. The best time is 118.62 miliseconds, achieved by padding the locks with 112 bytes, and the histograms with 128 bytes. Padding both buckets and locks hence reduces execution time by 71.8%. Stated differently: Without padding, the program takes 3.6 times as long to run as it does with padding. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the wall-clock execution time of the histogram problem, as a function of the amount of padding between the histogram buckets in bytes. Each y-tick is 100 milliseconds. The axes start at 0.

The `getAndIncrement` method performs the atomic compare-and-swap operation, and guarantees that updates are visible.

Reading the source code for the OpenJDK implementation of the `AtomicIntegerArray` class [18] reveals that the integers are stored in a plain `int` array. To ensure atomicity and visibility of updates, the implementation relies on the undocumented `sun.misc.Unsafe` class. This strongly suggests that our implementation suffers from false sharing of cache lines, as the array elements appear to be densely allocated. To confirm this suspicion, we run experiments with padding technique C.

Figures 9.18, 9.19, and 9.20 show that the execution time of this version of the histogram builder improves drastically when we introduce padding between the counters, confirming our suspicions that the `AtomicIntegerArray` class, and therefore our histogram builder, suffers from false sharing.

Platform	Best time (ms)	Padding (bytes)
i5	60.62	48
i7	25.26	128
Xeon	48.62	128

Figure 9.17: The best execution times for the compare-and-swap histogram builder, and the amounts of padding used to achieve them. Times are wall-clock execution times.

The best execution times, shown in figure 9.17, are better than those of the fine-grained locking implementation by factors of 1.9, 2.8, and 8.7 for the i5, i7, and Xeon platforms respectively. However, the communication-free implementation is still 3.3, 3.3, and 9.7 times faster than the compare-and-swap version.

The lesson here is that data-sharing between cores should be avoided entirely when doing so is easy. And when it cannot easily be avoided, we should at least avoid unnecessary communication caused by false sharing.

9.3.2 Quicksort

A famous example of a divide-and-conquer algorithm, Quicksort recursively divides the sorting problem into independent subproblems, combining partial results into a solution for the whole problem. We can see this problem division as the same kind of avoidance we saw in the communication-free histogram example: If the sub-problems are independent, they can be solved in parallel on a multicore system without any communication between parallel processes. Of course, we need to ensure that solutions are visible *after* they are found, but no communication is needed while the threads work. This is the only of our experiments where eliminating false sharing does not significantly improve execution times.

The experiments use a (somewhat naive) parallel implementation of quicksort, outlined in code snippet 9.10. Anytime the algorithm divides the problem in two, the calling thread forks a task to solve the left part of the problem, executing the right part itself. In this experiment we do not directly control the number of threads used on each platform: Forked tasks are run in parallel at the discretion of the `ForkJoinPool`.

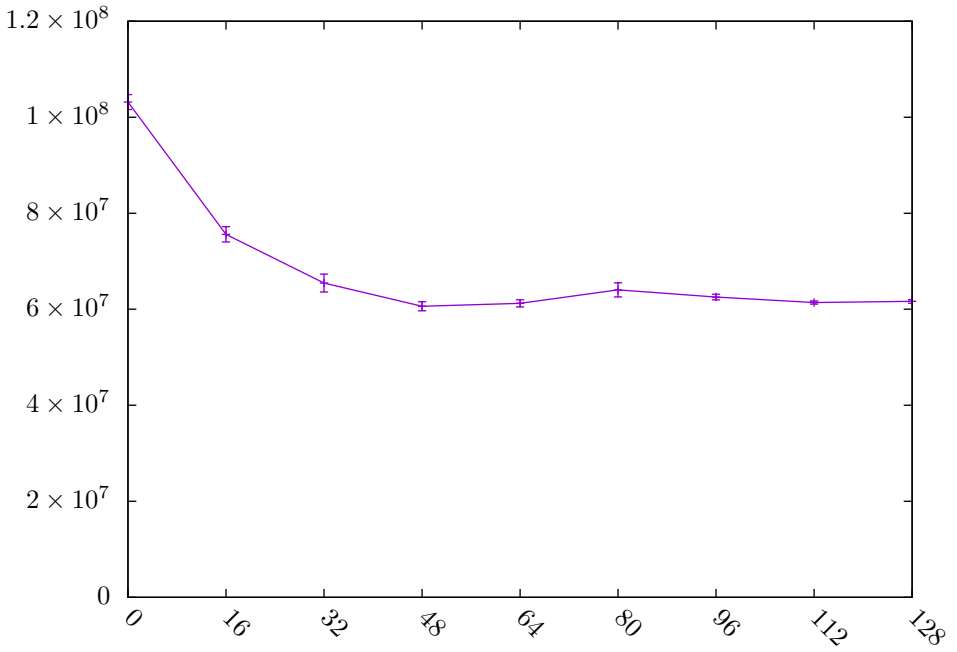


Figure 9.18: Execution times for the compare-and-swap based histogram builder on the i5 platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

```
public class QuickSort extends RecursiveAction {
  ...
  protected void compute() {
    if (hi-lo <= cutoff) {
      SelectionSort.sort(array,lo,hi);
      return;
    }
    int mid = partition();
    ForkJoinTask<Void> leftTask =
      new QuickSort(array, lo, mid, cutoff).fork();
    QuickSort right =
      new QuickSort(array, mid + 1, hi, cutoff);
    right.compute();
    leftTask.join();
  }
  ...
}
```

Code snippet 9.10: Simplified code for the Quicksort problem. The left-out `SelectionSort.sort` method implements sequential Selection sort. The left-out `partition` method implements a median-of-three version of Hoare partitioning.

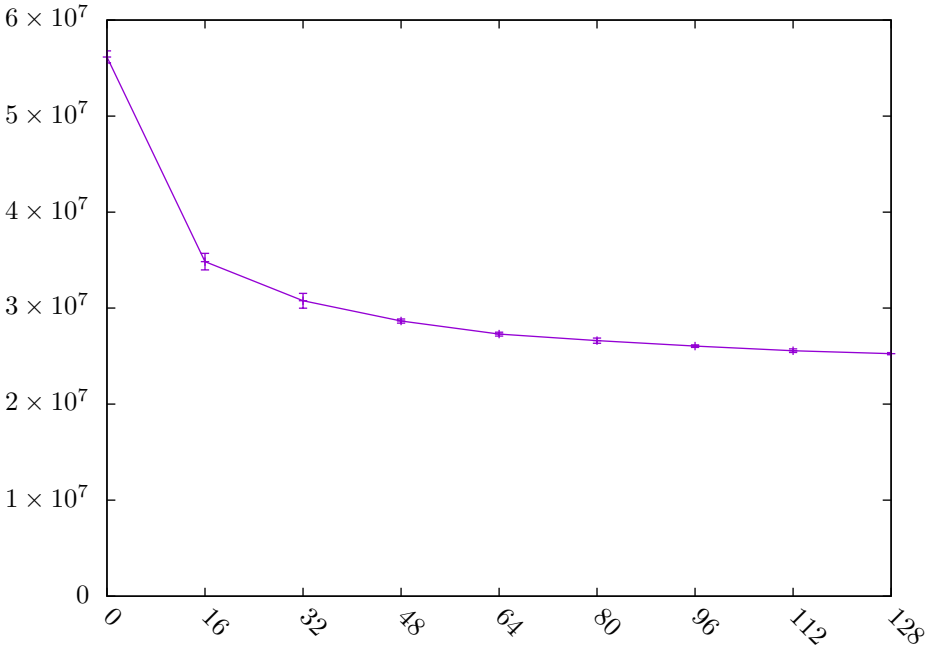


Figure 9.19: Execution times for the compare-and-swap based histogram builder on the i7 platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

As is common, and as is recommended in the popular algorithms text book by Sedgewick and Wayne [26], the implementation uses a cutoff to a sequential Selection sort. An experiment is included to find a good value for the cutoff.

There is no step to combine the results of two subproblems, as calls operate in-place on the same shared array. While this may *seem* like communication between threads, there is only a single thread operating on a given array-segment at any time. The only relevant communication is that of the constructor parameters to the recursive calls, and the reference to the array itself. Operations in recursive calls are guaranteed to be visible to the caller because of the call to `join()`.

There is a risk of false-sharing every time the algorithm subdivides the input array: The rightmost elements of the left part may share a cache line with the leftmost elements of the right part. We run three experiments on each platform:

padnone With no padding, used to determine a good value for the cutoff.

padall Where padding is added between all elements in the input array.

padsome Where padding is added between some elements in the input array, such that padding segments and data segments are of equal length.

We use padding technique C to space the array elements.

Experiments are run using an array with 4 million integers to be sorted. Padall and padsome experiments use a cutoff value of 16 array elements, corresponding to 64 bytes. Figures 9.21, 9.22, 9.23 show the execution times of

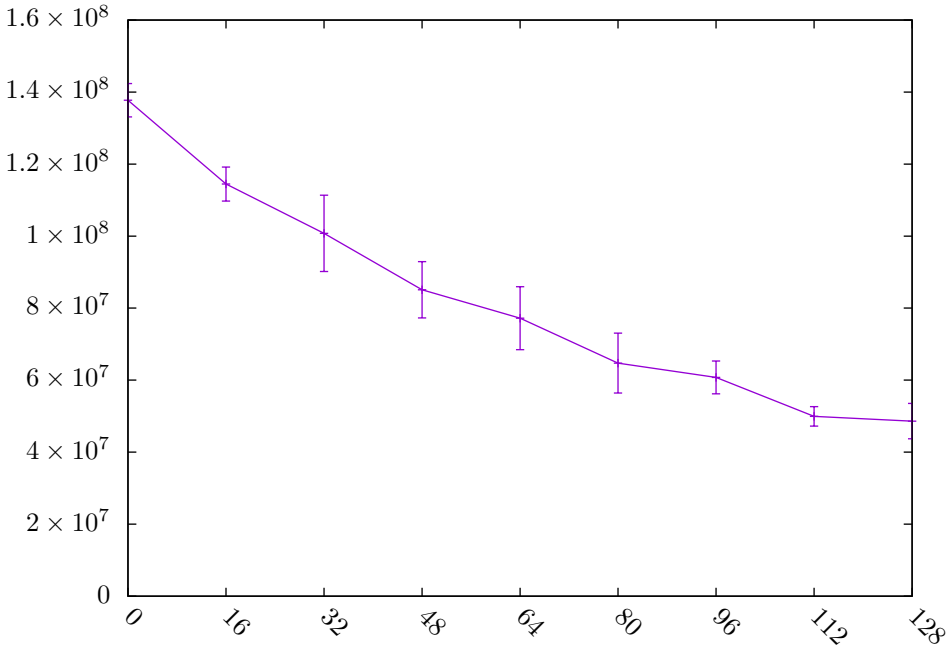


Figure 9.20: Execution times for the compare-and-swap based histogram builder on the Xeon platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

the experiments. The results show that adding padding between all array elements vastly increases execution time, while adding padding between just some elements decrease execution time by 0-1.8%

There are a few reasons why false sharing might not introduce significant communication overhead to the Quicksort implementation: At most one cache line can be falsely shared at each problem division. This means each task accesses at most 2 cache-lines that are falsely shared with other threads. An additional two cache lines per task may be subject to false invalidations due to the L2 prefetcher, as explained in section 9.3.1. Each task writes to each of their array elements at most once in the partitioning step. When performing Selection sort, each task performs up to $\frac{16 \cdot 17}{2} = 136$ writes, but all within the same 64-byte memory segment, so at most 2 cache lines are involved. As a result, the shared cache lines suffer contention, if any. Finally, nothing guarantees visibility of updates before a task has finished, so we do not suffer the full overhead of the cache coherence protocol.

The fact that we see slight improvement from padding in the padsome experiments might simply be noise, but it leads us to a new question. The experiments are all but guaranteed to have useless padding. That is, padding between elements that are always in the same subdivisions. With Quicksort, we cannot predict where the array will be divided beforehand, but perhaps other divide-and-conquer algorithms, such as Mergesort, could benefit more from eliminating false sharing, simply because we could add padding only in positions where the input array gets split.

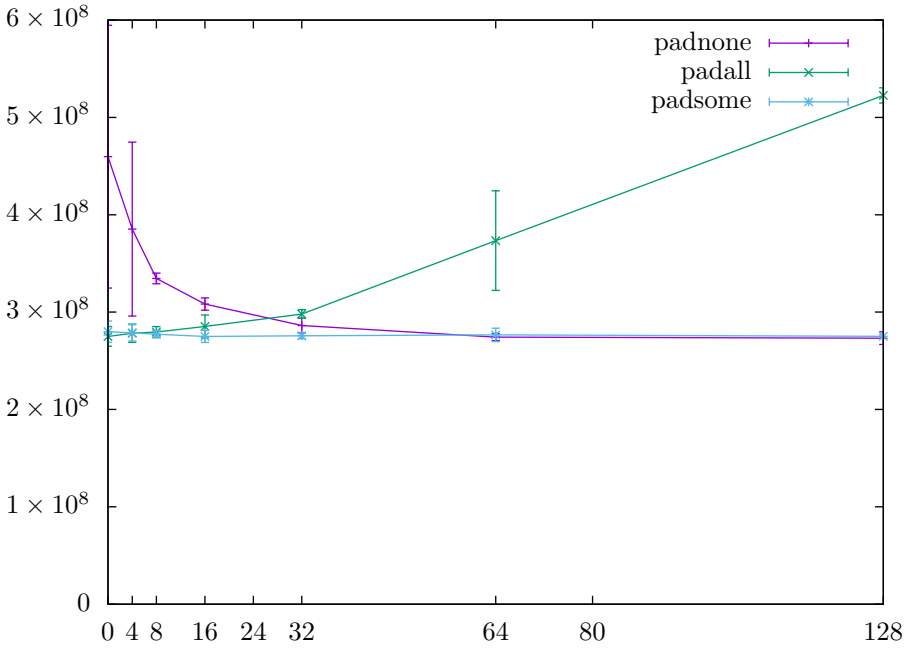


Figure 9.21: Quicksort on the i5 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

The plots show a high standard variations for some amounts of padding. On the Xeon platform in particular, the values for the padall experiment are highly unreliable. This may indicate that there is something wrong with my benchmarks, or that there is otherwise something more than meets the eye here.

9.3.3 K-means

The K-means problem from the 2017 PCPP exam – the original inspiration for *A multicore performance mystery solved*[3] and this report – is another example of a locking application which suffers from false sharing.

The k-means clustering algorithm takes as input: A list of points to be clustered, and a list of k initial cluster means. The algorithm then assigns each point to its nearest cluster, and updates the clusters' means according to the new assignments. This process is repeated iteratively until cluster means have stabilised.

All our k-means experiments are run with 200.000 points and 81 clusters, and take 108 iterations to complete.

The code used for this experiment is the same as in [3], with small adaptations, e.g. to the fields used for padding. The padding technique used is technique E. In fact, the example given for the technique in chapter 6 is taken from KMeans2Q64.

We examine 4 K-means implementations:

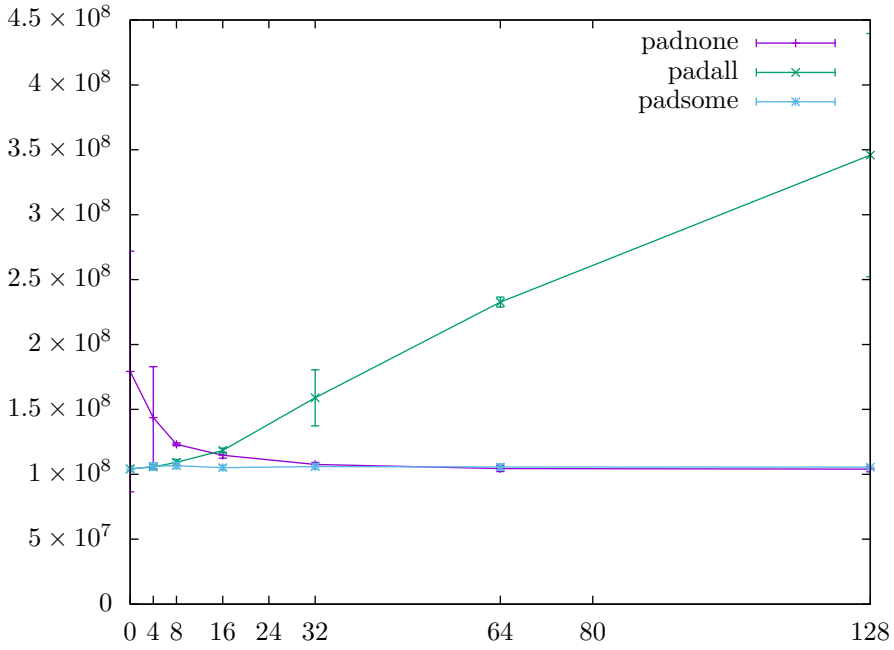


Figure 9.22: Quicksort on the i7 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

```
public static class Cluster implements Cluster{
  private volatile Point mean;
  private double sumx, sumy;
  private int count;
  public synchronized void addToMean(Point p) {
    sumx += p.x;
    sumy += p.y;
    count++;
  }
  public synchronized boolean computeNewMean() {
    ...
  }
```

Code snippet 9.11: Simplified code for the k-means `Cluster` class

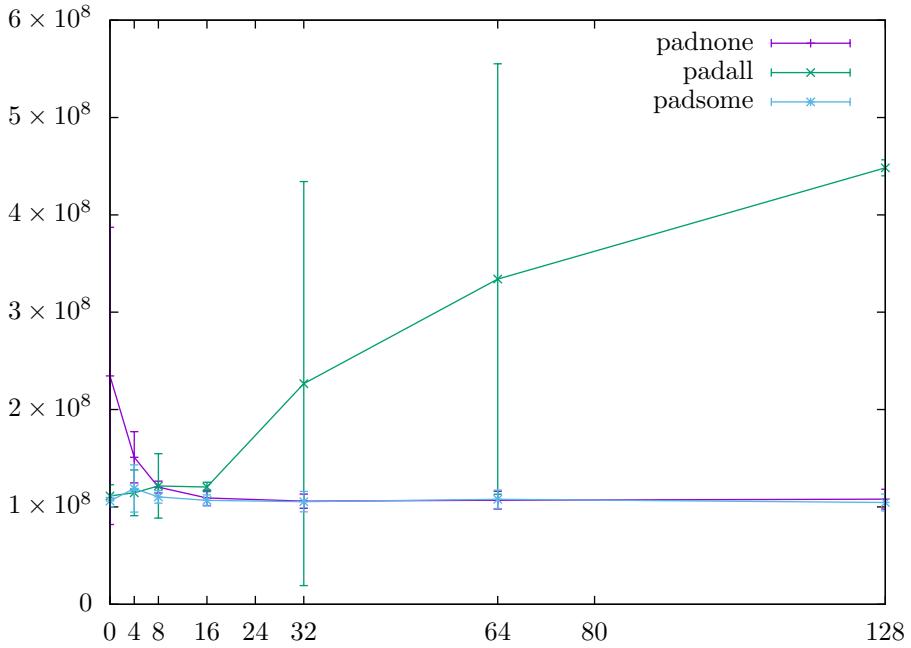


Figure 9.23: Quicksort on the Xeon platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

```
while (!converged) {
  // Assignment step: put each point in exactly one
  // cluster
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      myCluster[pi] = closest(points[pi], clusters);
  }
  // Update step: recompute mean of each cluster
  let taskCount parallel tasks do {
    for (int pi=from; pi<to; pi++)
      myCluster[pi].addToMean(points[pi]);
  }
  converged = true;
  for (NormalCluster c : clusters)
    converged &= c.computeNewMean();
}
```

Code snippet 9.12: Simplified code for the original k-means implementation, KMeans2P.

```
while (!converged) {
  // Assignment step: put each point in exactly one
  // cluster
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      closest(points[pi], clusters)
        .addToMean(points[pi]);
  }
  // Update step: recompute mean of each cluster
  converged = true;
  for (Cluster c : clusters)
    converged &= c.computeNewMean();
}
```

Code snippet 9.13: Simplified code for the optimized k-means implementation, KMeans2Q.

KMeans2P The unoptimized implementation described in[3] and outlined in code snippet 9.12.

KMeans2Q The optimized implementation described in [3] and outlined in code snippet 9.13.

KMeans2Q64 The same as KMeans2Q, but with a `Cluster` class with two 64-byte padding segments.

KMeans2Q128 The same as KMeans2Q, but with a `Cluster` class with two 128-byte padding segments.

The most significant false sharing overhead comes from the `sumx`, `sumy`, `count`, and `mean` fields in the `Cluster` class (Hereinafter, we shall refer to the former three of these fields as the *assignment fields*). To see the problem, we need to understand just 3 points:

1. Updates to the assignment and `mean` fields are subject to the cache coherence protocol: They are guarded by a lock (the surrounding `Cluster` instance), and the `mean` field is declared `volatile`. Writes to these fields will result in invalidation messages being sent to the other cores. Reads from `mean` will force the CPU core to process pending invalidations, which may result in later cache misses.

2. The `mean` fields might be in the same cache line as the assignment fields. The fields consist of an object pointer, two `doubles` and an `int`, taking up a total of 28 bytes. The small memory footprint even makes it possible for fields in two separate `Cluster` to share a cache line!

3. Every call to the `closest` method reads the `mean` fields of *every cluster*. Calls to the `closest` and `addToMean` methods are interleaved in

KMeans2Q, which means writes to the assignment fields happen concurrently with a lot of reads from the `volatile mean` field.

This means that every write to the assignment fields runs the risk of causing false cache misses on all other CPU cores, if the assignment fields falsely share the cache lines of `mean` fields.

The padded versions of KMeans2Q arrange the fields so that the assignment fields are together in a contiguous memory segment, with padding both before and after that segment. That should guarantee that the `mean` field is never in the same cache line as the three other fields. No padding is added between the `sumx`, `sumy`, and `count` fields. These three fields are always updated together, so we welcome the possibility of them being placed in the same cache line.

K-means version	Time (ms)	SD (%)
Kmeans2P	2591.73	1.57
Kmeans2Q	3275.63	1.68
Kmeans2Q64	4389.13	8.73
Kmeans2Q128	3035.41	1.00

Figure 9.24: The k-means problem on the i5 platform, using 4 tasks. Time is the total wall-clock execution time, the standard deviation (SD) is given as the percentage of the execution time.

K-means version	Time (ms)	SD (%)
Kmeans2P	974.63	0.32
Kmeans2Q	1935.06	13.40
Kmeans2Q64	1907.32	4.26
Kmeans2Q128	1221.12	2.22

Figure 9.25: The k-means problem on the i7 platform, using 8 tasks. Time is the total wall-clock execution time, the standard deviation (SD) is given as the percentage of the execution time.

K-means version	Time (ms)	SD (%)
Kmeans2P	934.69	2.89
Kmeans2Q	9475.56	12.29
Kmeans2Q64	9035.89	8.22
Kmeans2Q128	4503.75	9.78

Figure 9.26: The k-means problem on the Xeon platform, using 48 tasks. Time is the total wall-clock execution time, the standard deviation (SD) is given as the percentage of the execution time.

The results, shown in figures 9.24, 9.25, and 9.26, show that false sharing has a significant effect on our k-means implementations. Particularly on the Xeon platform, where the KMeans2Q128 is 2.1 times faster than KMeans2Q. Curiously, the execution time doesn't get as close to that of KMeans2P as in

Sestoft's experiments, even though both experiments use two 128-byte padding segments in the `Cluster` class.

9.3.4 Striped hashmaps

Another problem used in the PCPP course, concurrent hashmap data structures provide a more complex example of striped locking applications than the histogram builder we examined earlier: Hashmaps serve as general key-value stores, their sizes may change dynamically, and unlike histograms, reads and writes to hashmaps are often interleaved.

We consider two versions of the striped hashmap: Striped map, and striped-write map. Both versions are from the PCPP course, and both versions are examined in [3]. However, [3] examines only the striped-write map with respect to false sharing.

Both implementations follow the same overall strategy: Key-value pairs are stored in buckets, chosen by hashing they key. Buckets are implemented as linked lists, allowing a single bucket to hold multiple key-value pairs. This takes care of hash collisions. Each bucket is assigned to a stripe, and each stripe is associated with a single lock that guards operations on buckets in that stripe.

The code included here is minimal. The hashmap implementations are still used as exercises in the PCPP course, and I do not wish to deprive students of the satisfaction of completing them. The full implementations are fairly intricate, but to see the risk of false sharing overhead, we need only consider the data structures they use internally.

Inspired by the `java.util.concurrent.atomic.LoncAdder` class[2], we use a quasi thread-local counter as the stress pattern for our hashmap experiments. Each thread reads its own segment of the input, and stores the sum of the inputs in a shared hashmap. Each thread uses a unique thread-id as key for its own counter. Whether the counters are effectively thread-local depends on whether the thread-ids hash to buckets in the same stripe. Experiments are run with 32 stripes on all platforms, regardless of thread count. The input sequence consists of 33 million integers, divided evenly between threads.

Striped map

The striped map relies on locking for both reading and writing operations (such as `get` and `put`). This ensures mutual exclusion as well as visibility of updates.

The elements of the `locks` array pose a risk of false sharing: An `Object` instance takes up 16 bytes[3], which means we can fit up to four locks in a single 64-byte cache line. Taking or releasing a lock can then falsely invalidate three other locks in the caches of other CPU cores.

The `buckets` and `sizes` fields are also candidates for false sharing, but are unlikely to cause significant overhead.

Figures 9.27 and 9.28 show the wall-clock execution time when using the striped hashmap with different amounts of padding between the locks. The locks are padded with padding technique D.

[2]While the LongAdder class ostensibly works in the same way as this experiment, it does not use a hashmap to store the counters, and uses an intricate optimistic concurrency scheme for updates.

[3]In chapter 6, we saw that this is the case on our three platforms, but it depends on the

```
let taskCount parallel tasks do {
    final int threadId = ...;
    final int to = ..., from = ... ;
    map.put(threadId, 0);
    for(int j = from; j < to; ++j) {
      int a = inputSequence[j];
      map.put(threadId, map.get(threadId) + num);
    }
    threads.add(t);
  }
```

Code snippet 9.14: Simplified code for the quasi thread-local counter we use for the hashmap experiments.

```
public static class StripedMap<K,V> {
  private volatile ItemNode<K,V>[] buckets;
  private Object[] locks;
  private final int[] sizes;
  ...
}
```

Code snippet 9.15: The most significant fields in the StripedMap class.

```
public static class StripedWriteMap<K,V> {
  private volatile ItemNode<K,V>[] buckets;
  private Object[] locks;
  private AtomicIntegerArray sizes;
  ...
}
```

Code snippet 9.16: The most significant fields in the StripedWriteMap class.

Striped-write map

The striped-write map works in much the same way as the striped map, except it lowers lock contention by not taking locks for read operations like `get`. Visibility of writes is guaranteed by piggy-backing on the visibility guarantees of the `AtomicIntegerArray` class, used to store element-counts for each bucket. One disadvantage of this technique is that the `sizes` elements now effectively become `volatile`, causing a larger coherence overhead.

Figure 9.29 shows the wall-clock execution time when using the striped-write hashmap with different amounts of padding between the locks and elements of `sizes`. The locks are padded using padding technique D. The elements of sizes are padded by reusing the indirection array from the locks. This is equivalent to using padding technique C, but without the overhead of using the additional indirection array.

The experiments show that false sharing has a pronounced impact on both hashmaps across all three platforms, as simply padding elements of shared datastructures improve execution times. The biggest improvement is with the striped-write map on the Xeon platform. Here, using 112 bytes of padding reduces the execution time by 33%. The smallest improvement is with the striped hashmap on the i5 platform. Here, using 48 bytes of padding reduces the execution time by 17%.

There is an interesting artifact in the results: The striped-write map performs remarkably worse than the striped map in these experiments. This may be explained by its use of an immutable implementation of the `ItemNode` class: Each write allocates new nodes, incurring both the cost of the allocations and of garbage collecting the old nodes. A stress pattern with a higher ratio of read- to write write-operations (the ratio in our thread-local counter is $\sim 1/1$) should benefit from using the striped-write map, as the cost saved by not locking eclipses the cost of the extra allocations.

specific Java runtime platform.

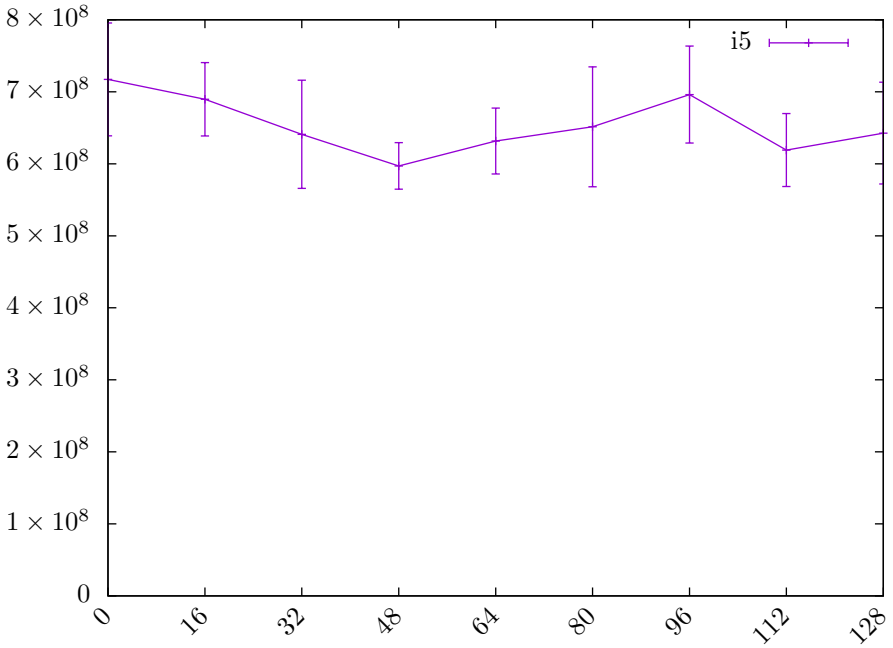


Figure 9.27: The striped hashmap problem on the i5 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

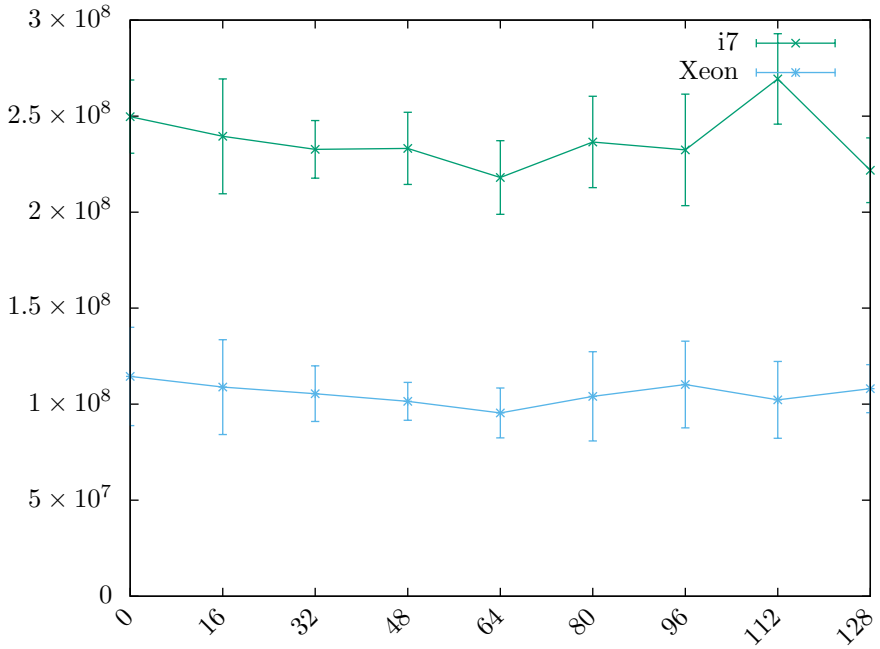


Figure 9.28: The striped hashmap problem on the i7 and Xeon platforms. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

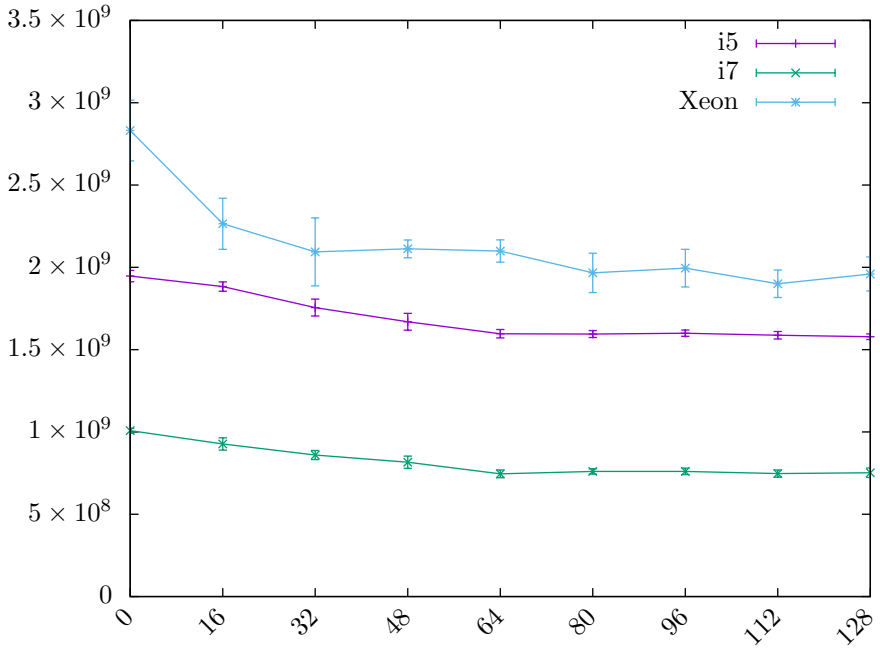


Figure 9.29: The striped-write hashmap problem on all 3 platforms. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

Chapter 10

Advice for multicore programmers

We have taken a look at cache behaviours, and seen that extra care needs to taken when writing cache-friendly multicore software. In particular, the cache coherence protocol means that grouping data closely together in memory, as we are wont to do in sequential software, can cause devastating false sharing overhead on multicore systems.

We have tried to relate the concepts of cache coherence and memory barriers and memory layout to Java, and have seen that steps can be taken to affect multicore cache performance in Java programs significantly:

We have seen from experiments that different multicore designs using locks or optimistic concurrency via compare-and-swap can be particularly sensitive to false sharing, and that using padded or spaced memory layouts can significantly improve performance of such applications. We have looked at designs that avoid sharing data between CPU cores by using thread-local data structures as the program runs, and only consolidating the results in end. When available, these designs perform remarkably well: They limit not only false sharing, but data sharing in general.

The lessons learned from this project are:

1. Sharing data between CPUs is expensive. So when practical, use thread-local designs to avoid sharing.

2. When avoidance is not practical, identify fields and data structures that are subject to updates, and use padding to ensure the updates do not cause false invalidations. This applies to all data, but is particularly important to synchronization variables like volatile fields, locks, and instances of the classes under the `java.util.concurrent` package.

3. False invalidation gets worse as the number of cores increase, but can be highly significant with just two physical cores.

4. All updates are candidates for false sharing overhead: Even if only a single thread performs writes, reading threads can force it to change a cache line's state from exclusive to shared. This forces the writing thread to perform – possibly false – invalidations for later updates.

5. Study the hardware. Spending a half an hour reading optimization manuals for the hardware platforms you use can replace hours of scouring the internet.

6. Benchmark multicore code. Studying hardware designs, while interesting and useful, is a rabbit hole of unbounded depth. Designing software with the cache coherence protocol in mind, we still ran into trouble with the L2 cache's spatial prefetcher with most locking applications. Or at least we think we did. We cannot know for certain, and that is exactly the point.

Bibliography

[1] Peter Sestoft. *Programming Language Concepts, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2017. ISBN 978-3-319-60788-7. doi: 10.1007/978-3-319-60789-4. URL https://doi.org/10.1007/978-3-319-60789-4.

[2] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10): 859–866, October 1972. ISSN 0001-0782. doi: 10.1145/355604.361591. URL http://doi.acm.org/10.1145/355604.361591.

[3] Peter Sestoft. A multicore performance mystery solved. 2017.

[4] Paul McKenney. *Is Parallel Programming Hard, And If So, What Can You Do About It?* 2015.

[5] Bjarne Stroustrup. C++11 style – a touch of class. https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style, January/February 2012.

[6] Ulrich Drepper. *What every Programmer Should Know About Memory*. 2007.

[7] Paul Mckenney. Memory barriers: a hardware view for software hackers. 08 2010.

[8] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[9] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984. ISSN 0163-5964. doi: 10.1145/773453.808204. URL http://doi.acm.org/10.1145/773453.808204.

[10] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1295480.1295483.

[11] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. *SIGARCH Comput. Archit. News*, 17

(2):257–270, April 1989. ISSN 0163-5964. doi: 10.1145/68182.68206. URL http://doi.acm.org/10.1145/68182.68206.

[12] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *SIGPLAN Not.*, 30(8):179–188, August 1995. ISSN 0362-1340. doi: 10.1145/209937.209955. URL http://doi.acm.org/10.1145/209937.209955.

[13] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *ICPP*, 1990.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055, 9780128119051.

[15] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201310090.

[16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java® language specification. URL https://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[17] Doug Lea and members of JCP JSR-166. Openjdk implementation of the atomicinteger class, . URL http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/atomic/AtomicInteger.java.

[18] Doug Lea and members of JCP JSR-166. Openjdk implementation of the atomicintegerarray class, . URL http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/atomic/AtomicIntegerArray.java.

[19] Doug Lea and members of JCP JSR-166. Openjdk implementation of the longadder class, . URL http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/atomic/LongAdder.java.

[20] Doug Lea and members of JCP JSR-166. Openjdk implementation of the striped64 class, . URL http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/atomic/Striped64.java.

[21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java® virtual machine specification. URL https://docs.oracle.com/javase/specs/jvms/se8/html/.

[22] Peter Sestoft. Microbenchmarks in java and c#. 2015.

[23] Peter Sestoft. Examination, practical concurrent and parallel programming, 10-11 january 2017. . URL http://www.itu.dk/people/sestoft/itu/PCPP/E2016/pcpp-20170110.pdf.

[24] Peter Sestoft. Supporting code for examination in practical concurrent and parallel programming, january 2017. . URL http://www.itu.dk/people/sestoft/itu/PCPP/E2016/pcpp-20170110-code.zip.

[25] Aleksey Shipilev. Rfr (s): Jep-142: Reduce cache contention on specified fields. URL http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.html. Message on the openjdk mailing list.

[26] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011. ISBN 032157351X, 9780321573513.