Multicore Performance Demystified

Hardware matters for Java developers

Sigurt Bladt Dinesen
sidi@itu.dk

Supervisor:
Peter Sestoft

August 12, 2018

Contents

Chapter 1

Abstract

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.

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*[1], 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. The intended audience is the graduate- or post-graduate level software designer who is perhaps used to working on high-level, managed platforms (such as Java), and who is unaccustomed to considerations of low-level hardware details, such as cache coherence.

Chapter 3

How to read this report

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. And if they wish, Java programmers may ignore even the existence of the memory hierarchy altogether, and simply think about objects and their relationships.

The purpose of this section 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.

{add diagram of main-memory → L3-L1 cache → cpu cores}

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 10, 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 impact 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 [2] 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 [2]) 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. 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.

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. Though it not always the case, we generally consider the word size to be the unit of memory operations.

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 [3], 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. {the "see that cache..." claim is so essential to the report that it should be stated earlier (maybe even abstract or introduction, otherwise just beginning of arch chapter), and repeated here.}

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 [4] 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 exact 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 we would have to store the memory addresses in addition to the cached values. Furthermore, any read or write operation must search the cache for 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 [4] [5].

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[6].

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. {It may be illustrative to show how to calculate the strides that cause this effect}

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 [4]. Intel's optimization reference-manual [6] 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.

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, not just sequential ones.

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[4]. 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

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.
{add diagram showing the same "variable" differing in main-mem/cpu0-cache/cpu1-cache/cpu1-registers}

This incoherence between what different cores see as the memory contents at a certain address, and the way it is solved in hardware, is what degrades performance in programs with false sharing.
{Continue writing: write-back caches (as described by drepper), MESI state states, store-buffers and invalidation queues (explanation of false sharing impact)} {mention that the unit here is technically a “block”. Cache line refers to a set/way pair, but cache-line and block are used interchangeably in literature}

Chapter 5

Background

{introduce section ("previous work")} In his paper [1] from 2017, Sestoft brings to attention the curious effects of false sharing of CPU 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 70% 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 parallel 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 parallel programs [7]. 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.

Like the loop fusion optimization that introduced the problem in [1], 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

[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.

data layout can go a long way. In [1], 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 [8] (the suite used was not the same as in [7]). 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. [9] 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 [1], 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 [8]

Their work [9] 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. [9] 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 [8], and 64 bytes in [1]).

For the purposes of this report, false sharing is defined as unrelated data being 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 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 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, and

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

- 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 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.

The definition used in this report most closely resembles what Bolosky and Scott calls "the hand tuning method", which they attribute to Jeremiassen and Eggers (referencing work not cited in this report).
{I haven't actually described bolosky and scotts methods yet}
{Write bg for the McKenney resources, and Drepper's what every programmer...} {Write bg for the (not very hardware-oriented) resources used at the outset of the project}

[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.

Chapter 6

Java and memory

{Explain relevant details of Java memory layout, to the point where the Method section can refer back here to say that we can't fully control the layout for our benchmarks} {Use and introduce the JOL tool} {I want to make sure that we ruminate a bit about dense allocation of locks here. It is not obvious that locking will write to the lock object, but it seems like it might be the case, and I don't want to clarify this in the experiments}

As noted previously, 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 garbage-collected languages 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, and the guarantees given by the Java Language Specification[11] and the classes in the `java.util.concurrent` package. The language specification gives no such guarantees regarding memory layout, so we have only limited control over it: 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, only we do not have guarantees that our optimizations will work on other Java runtimes, or with future Java versions.

The following sections explain briefly the relationship between Java's concurrency constructs and the memory hardware as described in chapter 4

6.1 Data sharing

In Java, the concept of *happens-before order* most closely captures the notion 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 [5]: 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. There is also the additional memory overhead of storing the object pointers in the array. Additionally, there is the performance cost of an added indirect; as accessing the elements now requires reading the object pointer from the array, and *then* reading the object.

The guarantees provided by the Java memory model do not correspond directly to those we know from hardware either. 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 [5], it is not clear why threads reading a different `volatile` field do not get the same guarantee.

Nonetheless, the remainder of this report will show that the hardware view taken in [5] 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[11, chapter 17][12, 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[12] 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[problemLength];
```

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 [1], as well as our own experiments in chapter 10, 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. a 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.

```
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 them.

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.

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 prefetch mechanisms.

Apart from better readability, I prefer this padding techniques 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 is not necessarily 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 loading the 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 following output:

Chapter 7

Memory operations are expensive

{(may be “the new FLOPS”)}
{Refer to bench-marks: (cyclic) Read-times vs simple math operation}

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 10. 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 3072KiB 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 3072KiB 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, end between the OpenJDK and Oracle platforms, but it lets us see that experiments are run on the Oracle Java(TM) SE Runtime Environment, version 1.8.0_144.

Benchmarks are run using the `mark8` method from Sestoft's microbenchmarking framework [13], 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. 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 10 we measure the idle CPU load of each platform, to convince ourselves that the noise levels are insignificant.

Chapter 9

The types of “time waste” in multicore programming

{“Cache-friendly” gets a new meaning}

Chapter 10

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, 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.

Unless otherwise noted, experiments are run with 4, 8, and 48 threads on the i5, i7, and Xeon platforms, respectively.

10.1 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 level 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.

10.1.1 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 10.1 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 loads 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.

10.1.2 Degrees of parallelism

```
let taskCount parallel tasks do {
  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 10.1: Simplified code for the memory-bound parallelism experiment.

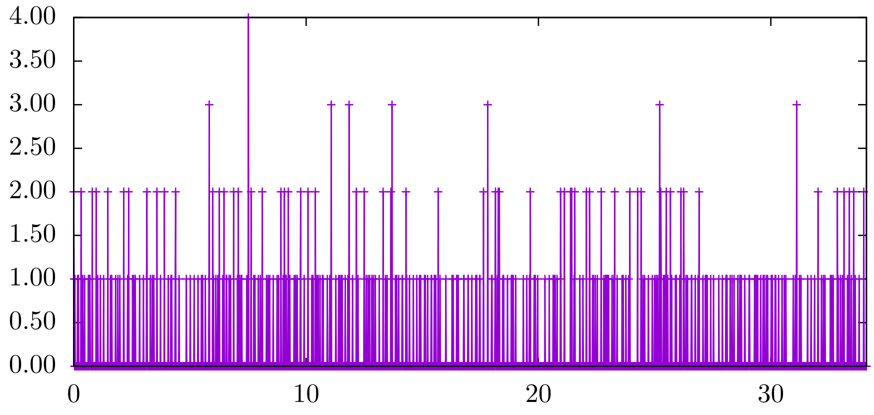
```
let taskCount parallel tasks do {
  double d = Double.MAX_VALUE;
  for (long j = 0; j < workPerThread; ++j) {
    d /= divisor;
  }
}
```

Code snippet 10.2: Simplified code for the CPU-bound parallelism experiment.

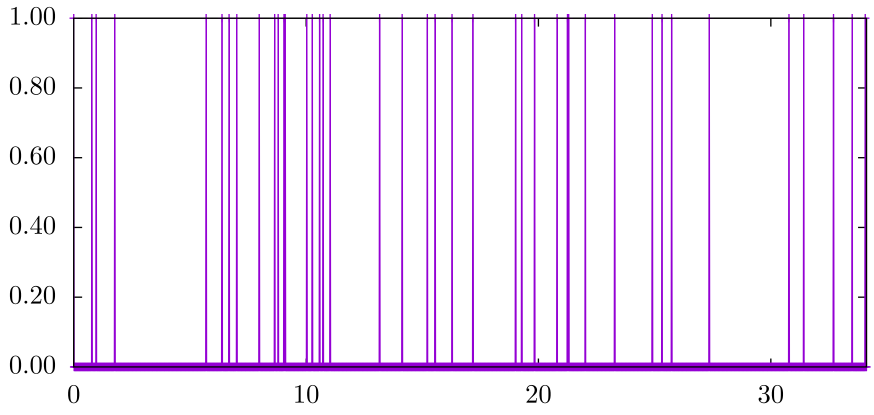
10.1.3 Memory hierarchy access-times

10.2 Multicore cache performance

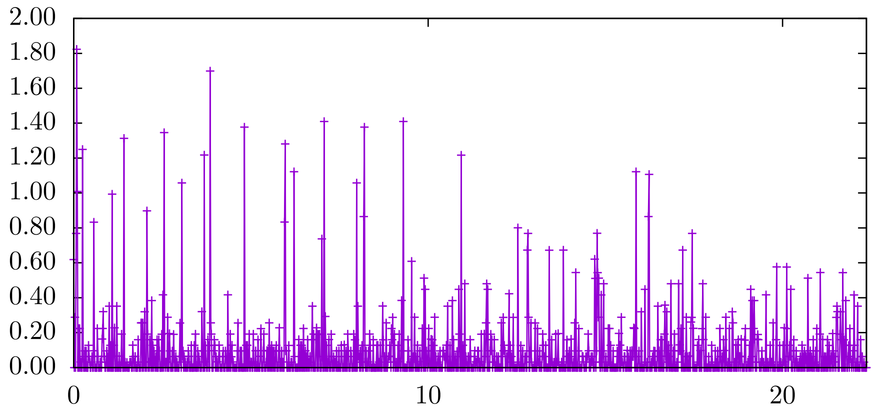
To see the possible impact of false sharing, it is illustrative to look at a few contrived example programs. In this section we look at a handful of variations of programs that perform simple operations in a multicore setting.



(a) i5



(b) i7



(c) Xeon

Figure 10.1: 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.

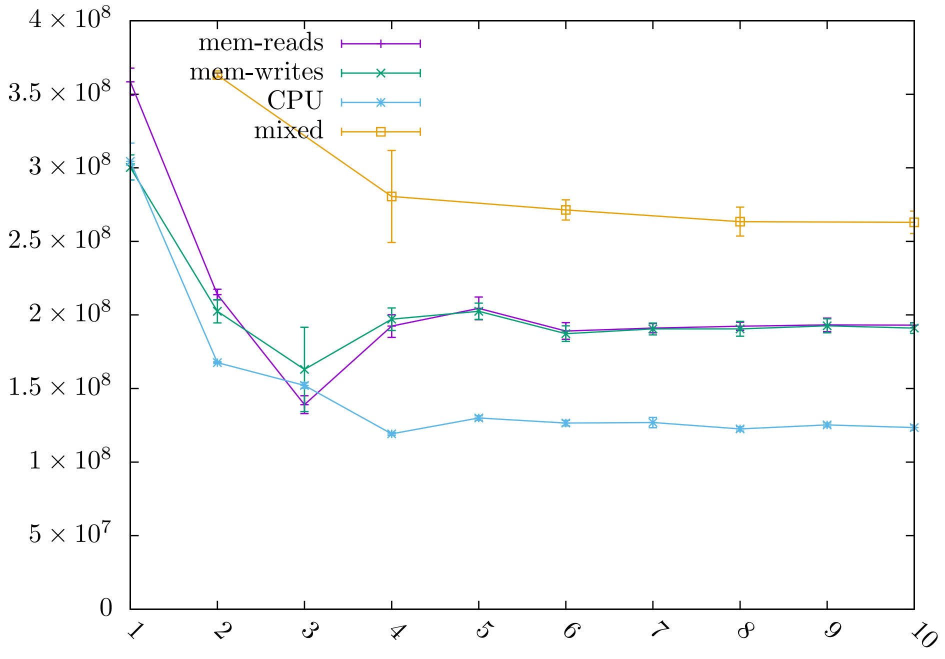


Figure 10.2: Parallelism experiment on the i5 platform. The plot shows wall-clock execution time in ns., as a function of thread count.

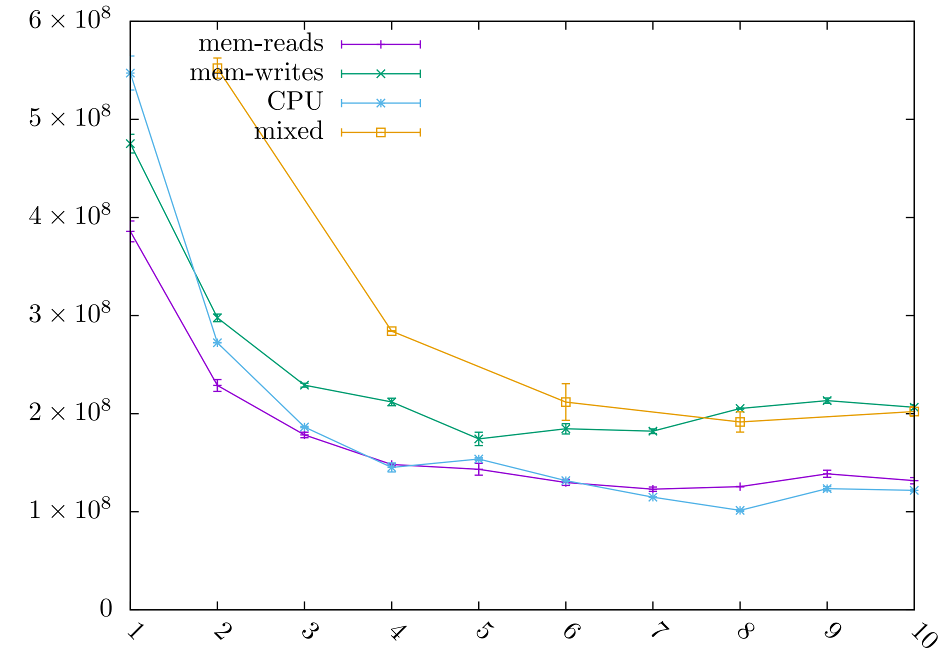


Figure 10.3: Parallelism experiment on the i7 platform. The plot shows wall-clock execution time in ns., as a function of thread count.

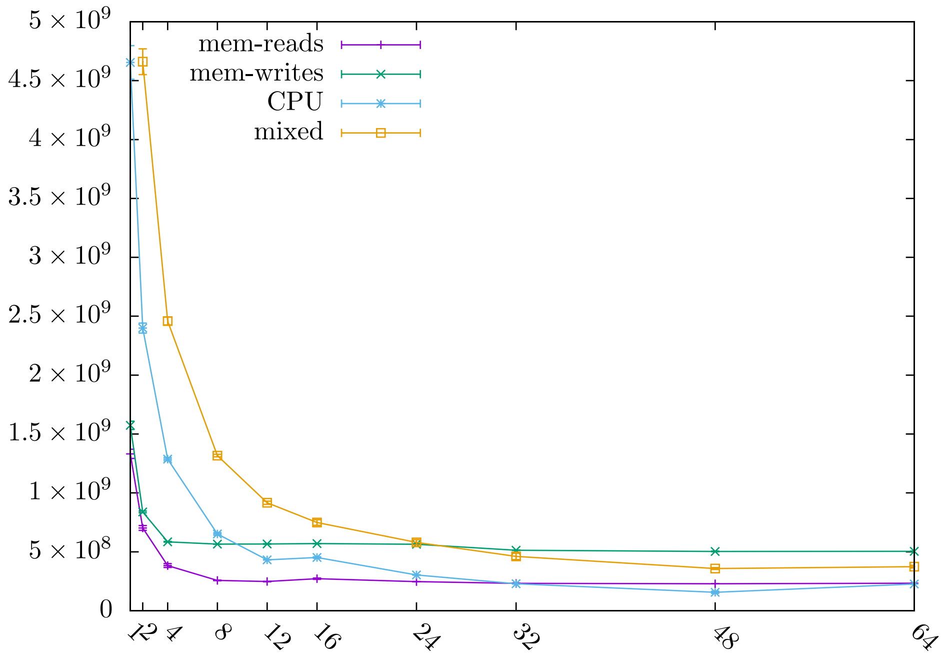


Figure 10.4: Parallelism experiment on the Xeon platform. The plot shows wall-clock execution time in ns., as a function of thread count.

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

Code snippet 10.3: Code for measuring memory access times.

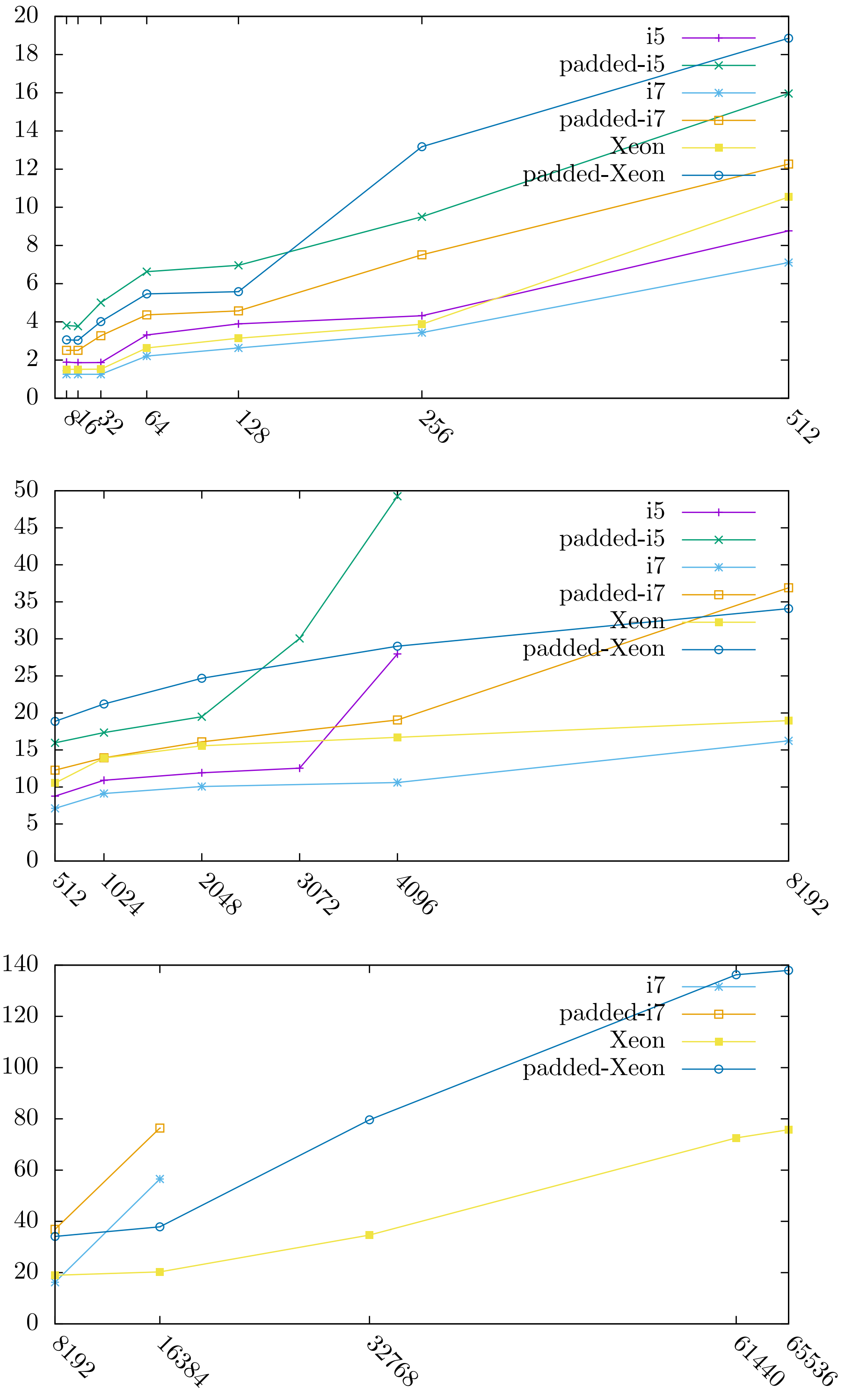


Figure 10.5: Random cyclic reads, approximating memory-access times on all 3 platforms. The y-axes show the average time per read operation, measured over 2^{25} reads. The x-axes show the working-set size in KiB.

10.2.1 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 10.4: Simplified code for the local-field version of the uncontended-writes experiment.

Each thread performs millions of increments. 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. Since each thread operates on its own field, there is 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 fiends 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 integers, each thread performs 6M increments. In the experiments without volatile, each thread performs 66M increments. Figure 10.6 and 10.7 show the average time per increment, as a function of padding.
{Add explanation of the different plots (array/local, barrier/no-barrier), include values of plots (plots are hard to read), and reflect/conclude}

10.2.2 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.

The program is similar to {reference uncontended snippet}, 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 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 allows us to see that, perhaps contrary to intuition, read operations in a multicore setting also incur a coherency overhead.
{include getter/stop code snippet}

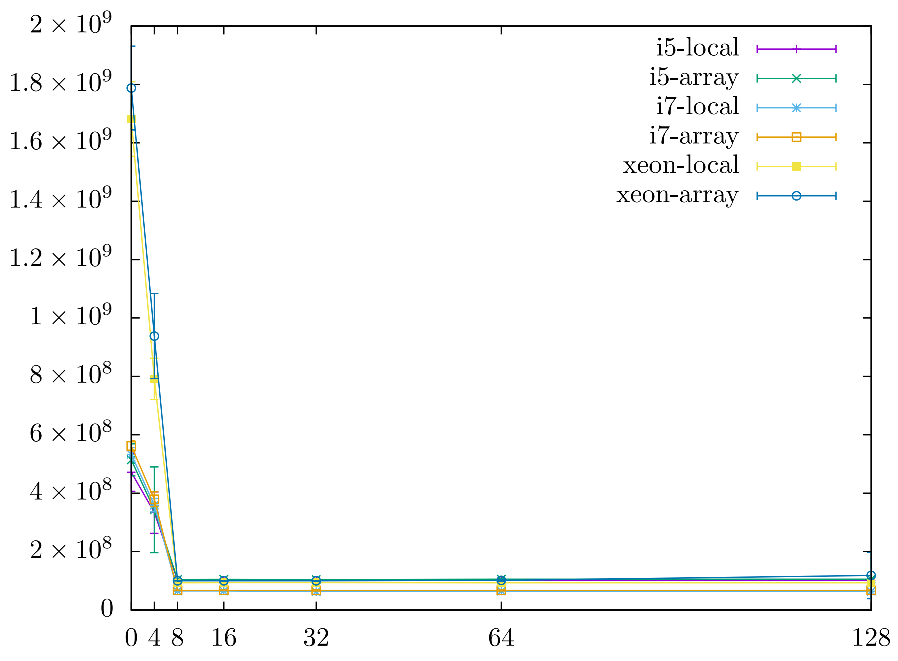


Figure 10.6: Uncontended increments on volatile integers. The plot shows wall-clock execution time, as a function of the number of bytes used as padding between the integers.

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

Code snippet 10.5: Simplified code for the local-field version of the contended-writes experiment.

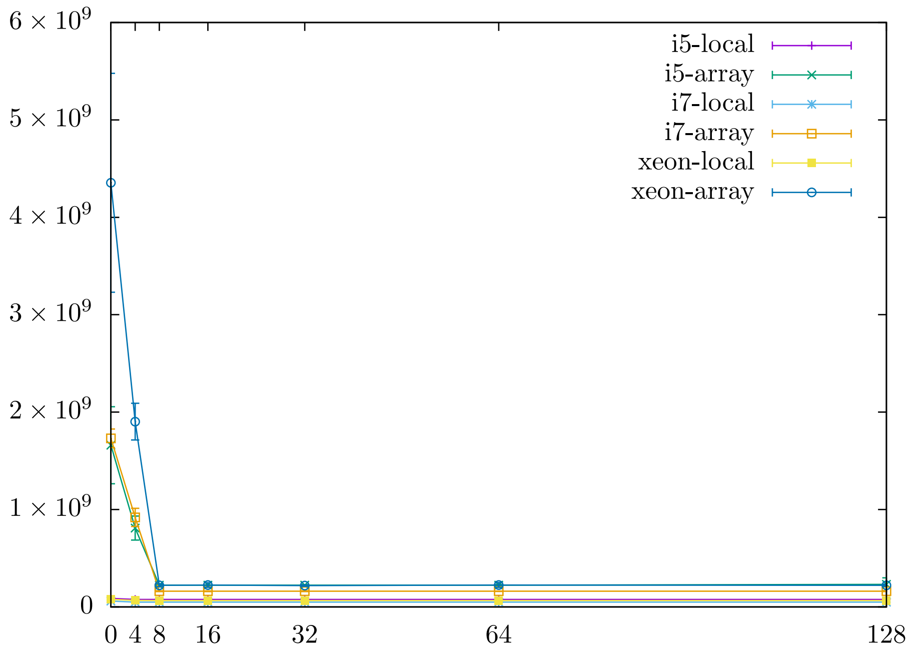


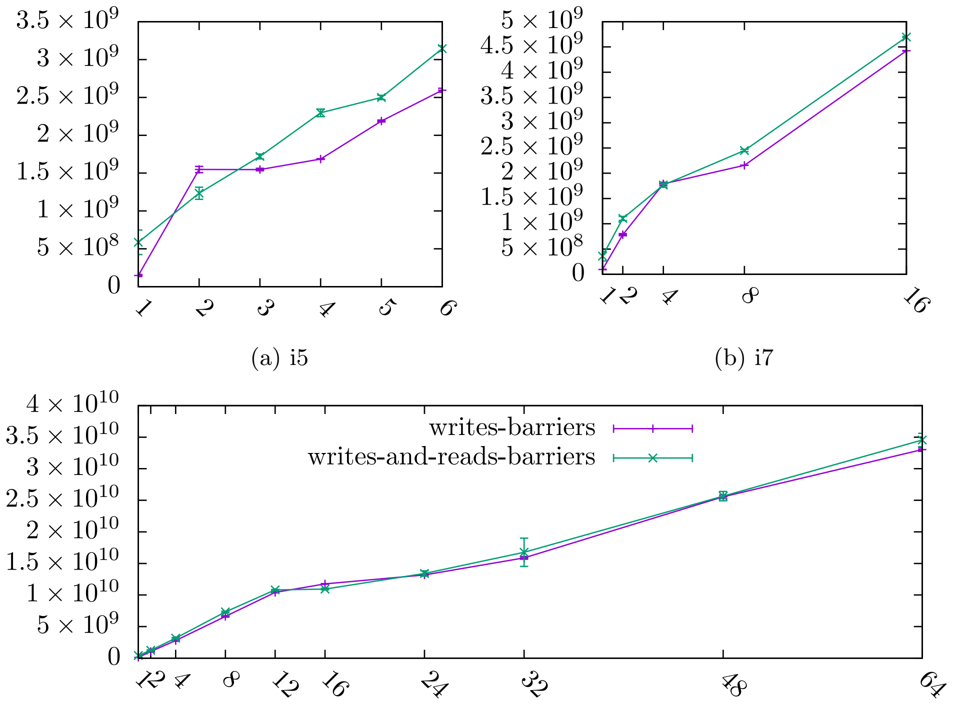
Figure 10.7: Uncontended increments on non-volatile integers. The plot shows wall-clock execution time, as a function of the number of bytes used as padding between the integers.

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 has shown that the time it takes to stop a thread is far too small to significantly skew our results. {include table of numbers (they are 833.4±2.82 (desktop), 1940.5±58.12 (server), and 1256.7±13.59 (laptop) ns)})
{include plots as ns-per-operation, concrete numbers/factors, and discussion/-conclusion on the results, concrete parameters: work not divided..}
{Konklusioner: Læsning er ikke gratis(!), pris for contended writes (både med og uden barrier/volatile) siger en del om MESI pris (1 vs 2 tråde er illustrativt), indikerer (løst) pris af at smide en cache-line frem og tilbage mellem kerner}

10.3 Practical applications

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

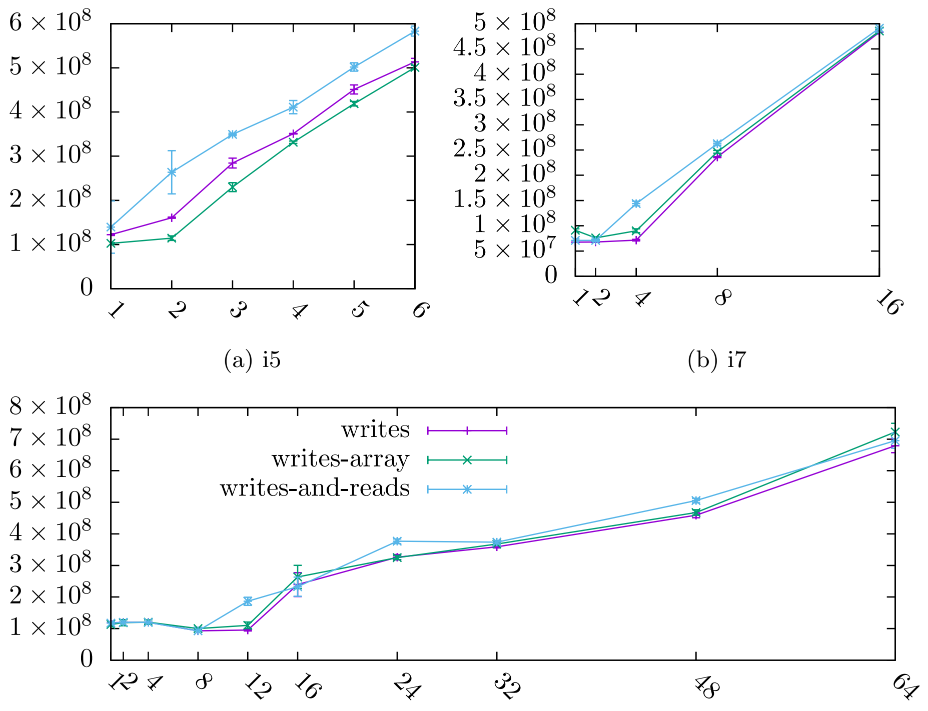


(a) i5

(b) i7

(c) Xeon

Figure 10.8: Contended increments on **volatile** integers. The plots show nanoseconds per increment, as a function of the thread count. Please note that both axes vary across the plots.



(a) i5

(b) i7

(c) Xeon

Figure 10.9: Contended increments on **non-volatile** integers. The plots show nanoseconds per increment, as a function of the number of threads. Please note that both axes vary across the plots.

sharing in lock-free compare-and-swap based applications. 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 [14, 15], as well as two implementations of a concurrent hashmap data structure used in the same course. Both of these applications are studied in [1], and can be said to have instigated this project.

10.3.1 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.

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: 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 10.6 outlines the implementation of this approach.

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

Figure 10.10: 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.

```
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 10.6: 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. Only reading from the input sequence is parallelised. At any time, only one thread can be incrementing a counter.

```
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 10.7: Simplified code for the threads in the coarse-grained locking version of the histogram builder.

The table in figure 10.11 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 10.11: 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 not 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 all their work in parallel, except when different threads work on counters guarded by the same lock. For practical applications the ideal level of granularity should be determined by experiment. In this experiment I use 32 locks: One lock for each bucket.

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 any actual work.

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.

```
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 10.8: Simplified code for the threads in the fine-grained locking version of the histogram builder.

We measure the impact of false sharing by benchmarking versions with different amounts of padding between the locks and counters.
{Because the elements/locks are objects, there is an additional, unadvertised, 12 bytes of padding used for the counters, and something similar for the locks (I can't say where in the object header the actual lock is, if it is even in the object header). This should e noted somewhere, but probably not here, as it goes for most of the benchmarks.}

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 10.12: 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.

Figures 10.13, 10.14, and 10.15 show the wall-clock execution times using different amounts of padding for the i5, i7, and Xeon platforms respectively. Each smaller plot shows the execution time of the histogram builder, as a function of the amount of padding between the counters. The third axis – the amount of padding between the locks – is unrolled into different plots.

The table in figure 10.12 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.

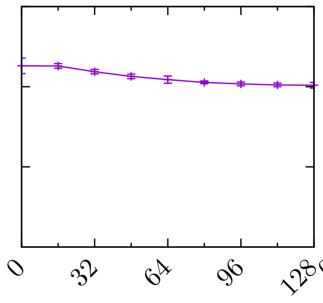
It is not surprising that the platform with the most cores suffers the worst. The more threads running in parallel, the bigger the likelihood of invalidating a cache line that is relevant to another core. Furthermore, the invalidation messages must be sent to all the other cores, and processors must wait for acknowledgement messages from all the 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.

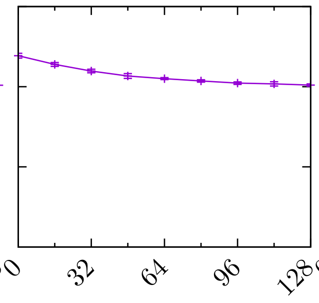
We would expect the performance to stop improving abruptly around 64 bytes, as that is the cache line size used by our hardware architectures. One explanation for the continued improvements is the cache prefetch mechanism. It is possible that the prefetch mechanism causes more than a single cache line 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[16]. This explanation is in agreement with the Intel optimization manual[6], 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.

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.

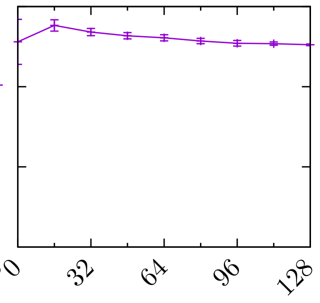
[1]It is not entirely clear from the manual which microarchitectures this applies to. It is stated for the Sandy Bridge microarchitecture, and seems to apply to those that succeed it as well.



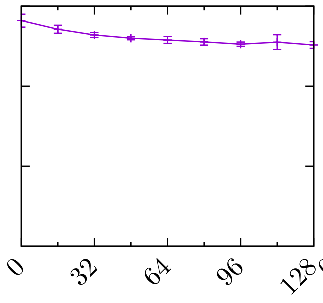
(a) 0 bytes



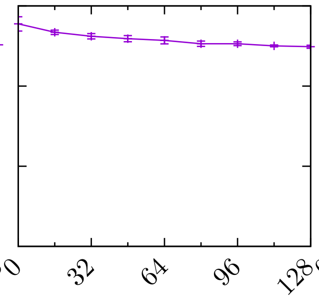
(b) 16 bytes



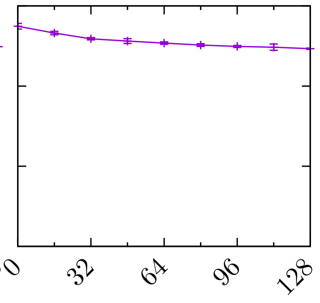
(c) 32 bytes



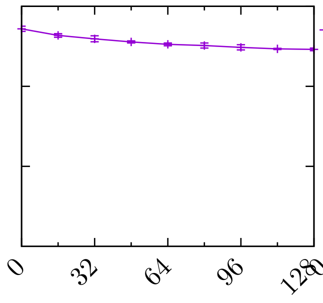
(d) 48 bytes



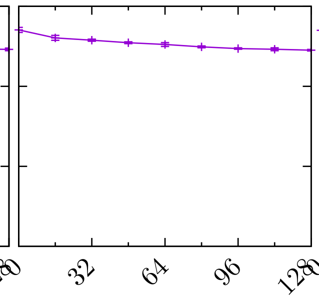
(e) 64 bytes



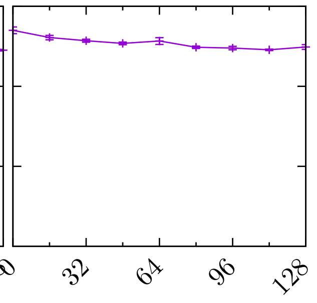
(f) 80 bytes



(g) 96 bytes



(h) 112 bytes



(i) 128 bytes

Figure 10.13: 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.

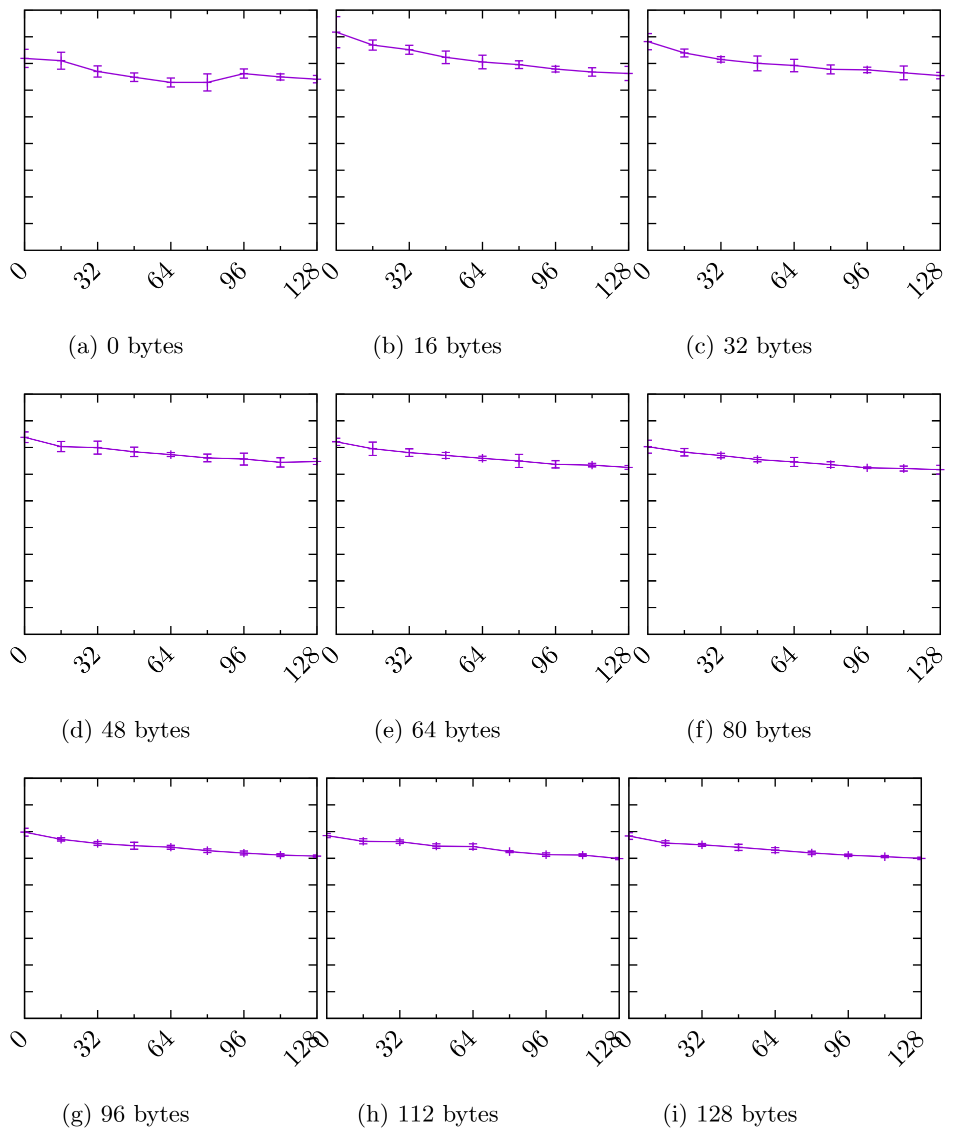
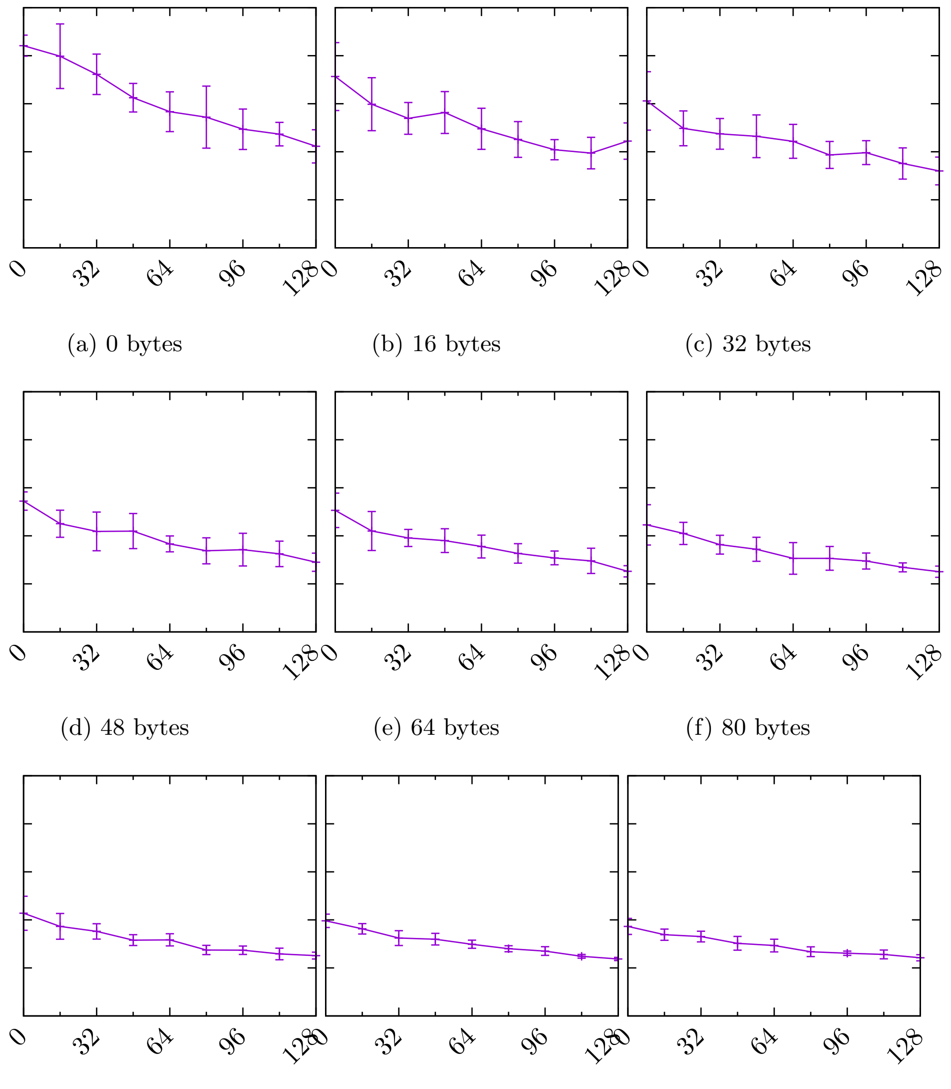


Figure 10.14: The fine-grained histogram problem on the i7 platform. Here the overall trend is that padding improves performance with respect to both buckets and locks. The best time is 59.9 milliseconds, achieved by padding the locks with 112 bytes, and the buckets with 128 bytes. Padding both buckets 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 10.15: 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.

Compare-and-swap

Following the same strategy as the fine-grained locking version implementation above, 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 contended. 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 locks 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 10.9: Simplified code for the threads in the compare-and-swap version of the histogram builder.

Code snippet 10.9 outlines the compare-and-swap implementation of the histogram builder. It uses Java's `AtomicIntegerArray` class to hold counters. The `getAndIncrement` method performs the compare-and-swap operation, and guarantees that updates are visible.

Reading the source code for the OpenJDK implementation of the `AtomicIntegerArray` class [17] 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 unpadded.

Figures 10.17, 10.18, and 10.19 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 10.16: 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 10.16, 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.

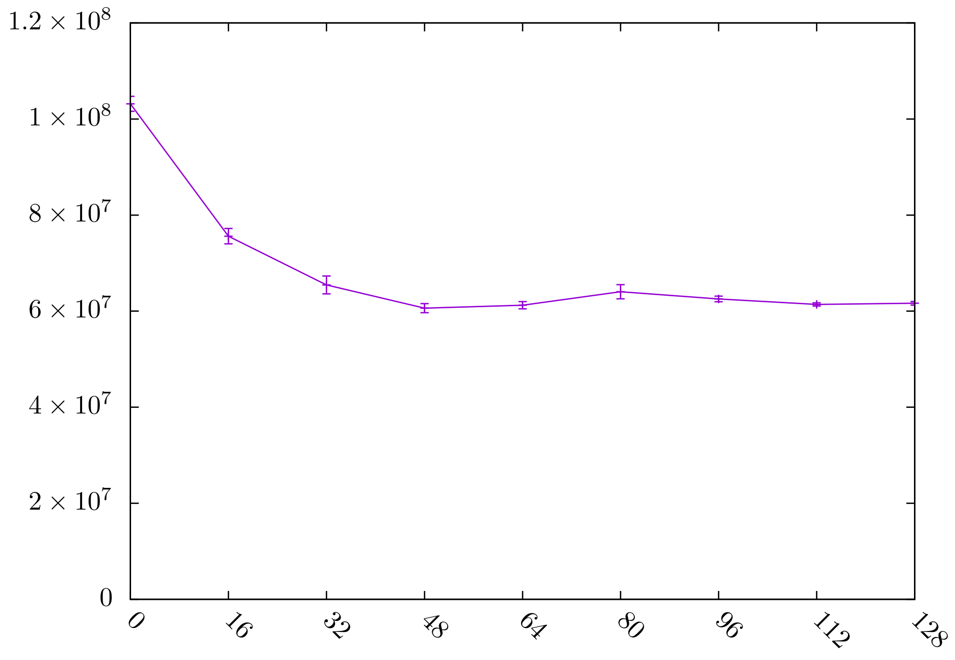


Figure 10.17: Execution times for the compare-and-swap based histogram builder on the i5 platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

10.3.2 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 whare eliminating false sharing does not significantly improve execution times.

The experiments use a (somewhat naive) parallel implementation of quicksort, outlined in code snippet 10.10. Anytime the algorithm divides the problem in two, the calling thread forks a task to solve the left part of the problem, ex-

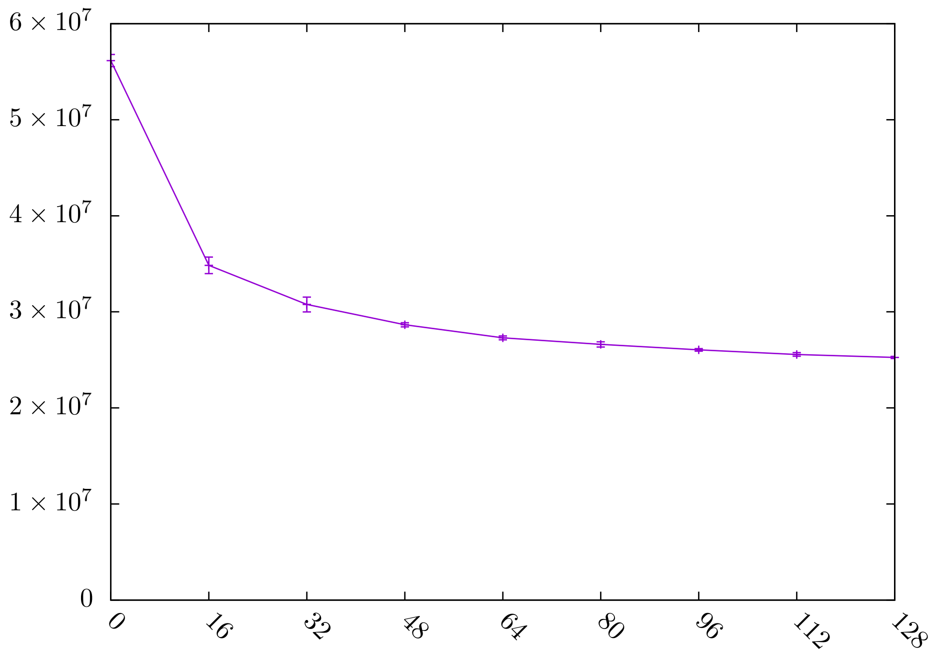


Figure 10.18: Execution times for the compare-and-swap based histogram builder on the i7 platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

```
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 10.10: 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.

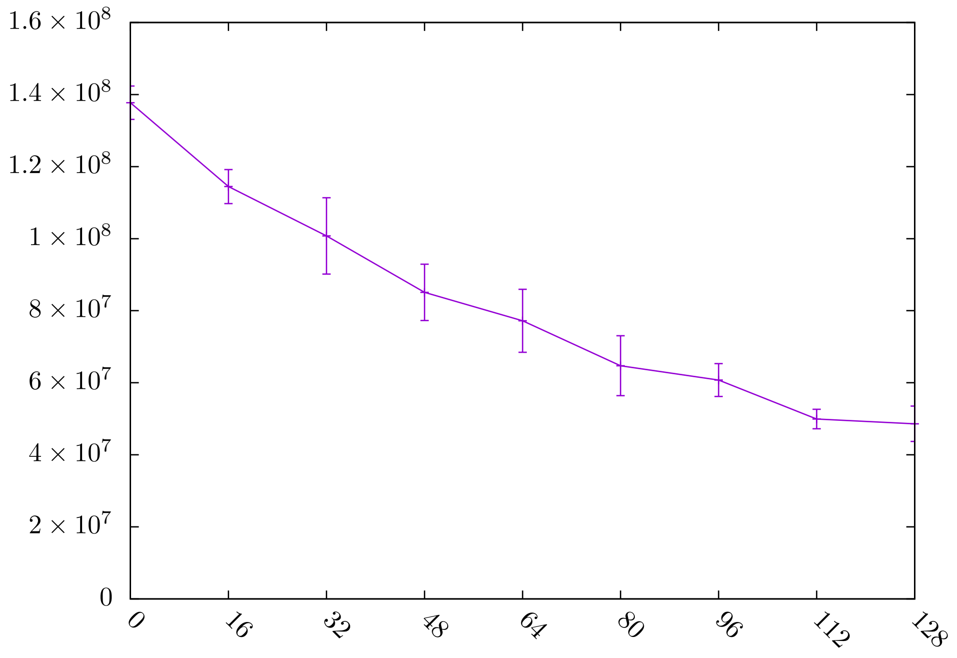


Figure 10.19: Execution times for the compare-and-swap based histogram builder on the Xeon platform. The plot shows the wall-clock execution time in ns., as a function of padding in bytes.

ecuting 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`.

As is common, and as is recommended in the popular algorithms text book by Sedgewick and Wayne [18], 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.

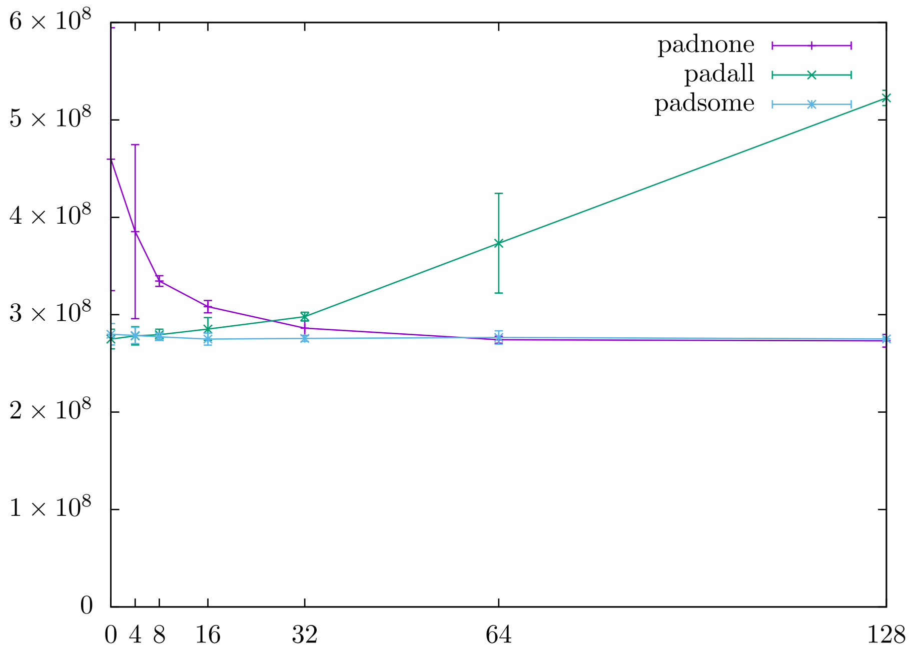


Figure 10.20: Quicksort on the i5 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

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 10.20, 10.21, 10.22 show the execution times of 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 10.3.1. Each task writes to each of their array elements at most once, either in the partitioning step, or when performing Selection sort, so the shared cache lines suffer little 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

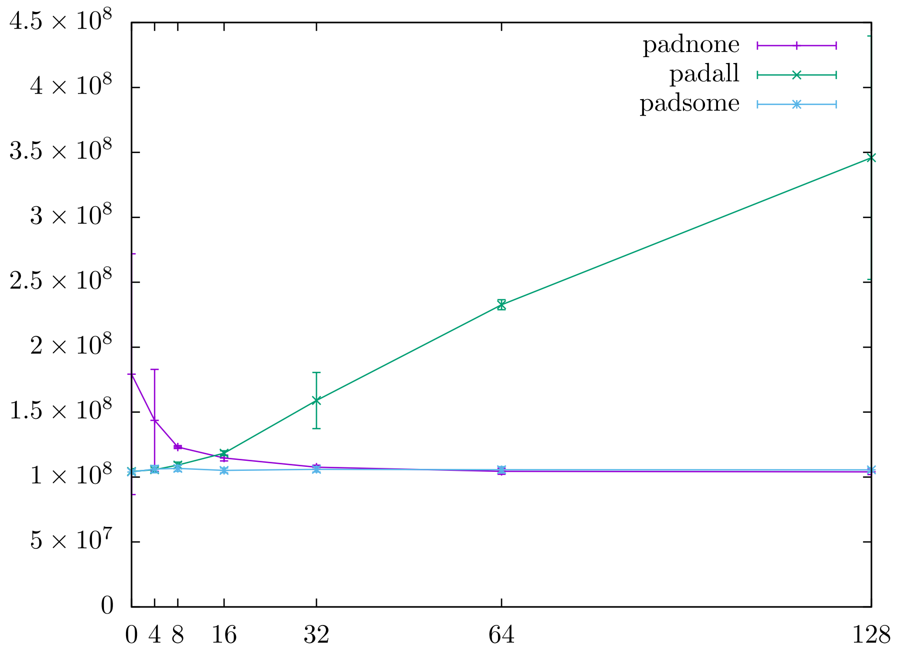


Figure 10.21: Quicksort on the i7 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

input array gets split.

10.3.3 K-means

The K-means problem from the 2017 PCPP exam – the original inspiration for *A multicore performance mystery solved*[1] 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 [1], with small adaptations. We examine 4 K-means implementations:

KMeans2P The unoptimized implementation described in[1] and outlined in code snippet 10.12.

KMeans2Q The optimized implementation described in [1] and outlined in code snippet 10.13.

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

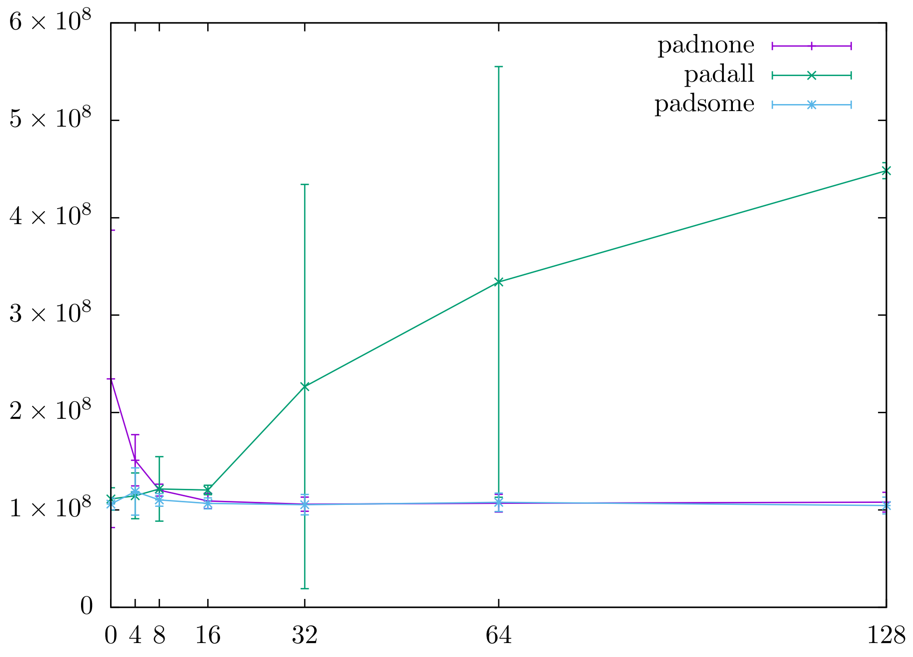


Figure 10.22: Quicksort on the Xeon platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes. For the padnone experiment, the x-axis indicates the cutoff value in bytes.

```
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 10.11: Simplified code for the k-means `Cluster` class

```
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 10.12: 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 10.13: Simplified code for the optimized k-means implementation, KMeans2Q.

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 these 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 guarded by a lock (the surrounding `Cluster` instance), and the `mean` field is declared `volatile`. This means that the fields are subject to the cache coherence protocol: 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 two separate `Cluster` instances to have fields in the same 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 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 10.23: 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.

The results, shown in figures 10.23, 10.24, and 10.25, 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.

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 10.24: 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 10.25: 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.

10.3.4 Striped hashmap

Another problem used in the PCPP course, concurrent hashmap data structures provide a more interesting 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 [1]. Howver, [1] 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.

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.

```
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 10.14: 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 10.15: The most significant fields in the StripedMap class.

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[2], 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: Writes and reads to these fields are protected by locks, so the contention of these fields is bounded, and likely overshadowed, by the contention of the locks.

Figures 10.26 and 10.27 show the wall-clock execution time when using the striped hashmap with different amounts of padding between the locks.

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.

[2]In chapter 6, we saw that this is the case on our three platforms, but it depends on the specific Java runtime platform.

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

Code snippet 10.16: The most significant fields in the StripedWriteMap class.

One disadvantage of this technique is that the `sizes` elements now effectively become `volatile`, causing a larger coherence overhead.

Figure 10.28 shows the wall-clock execution time when using the striped-write hashmap with different amounts of padding between the locks.

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.

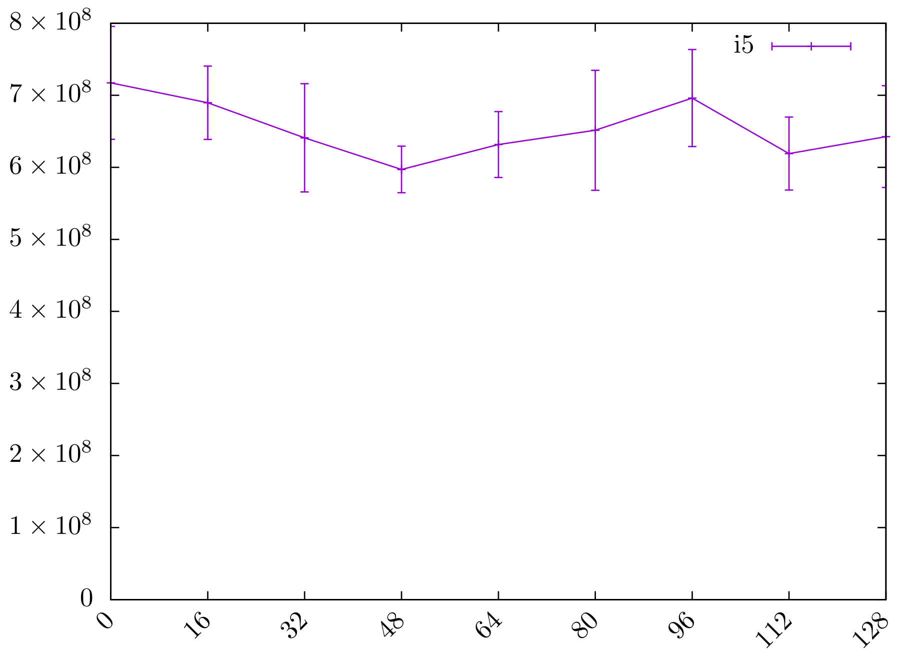


Figure 10.26: The striped hashmap problem on the i5 platform. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

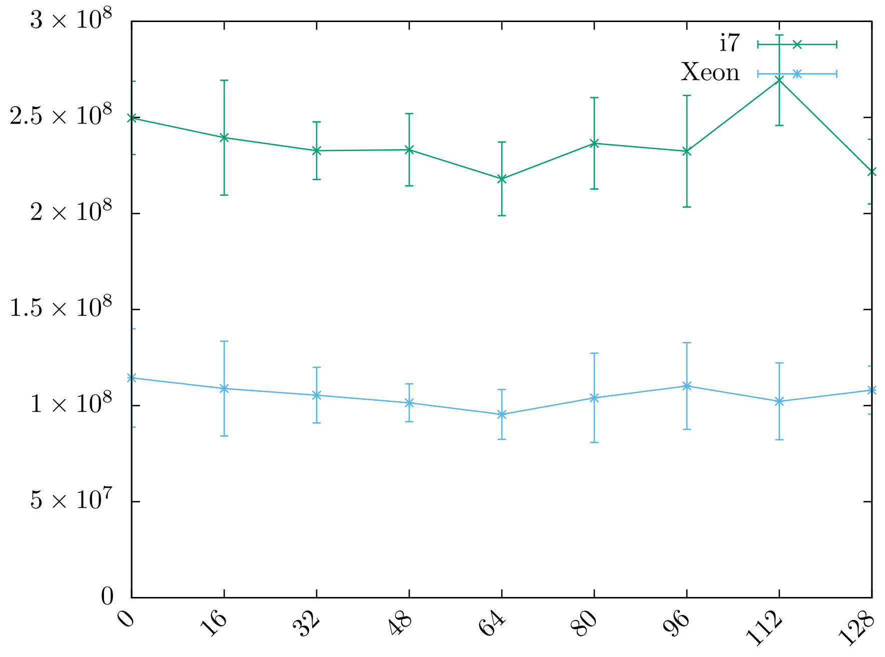


Figure 10.27: The striped hashmap problem on the i7 and Xeon platforms. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

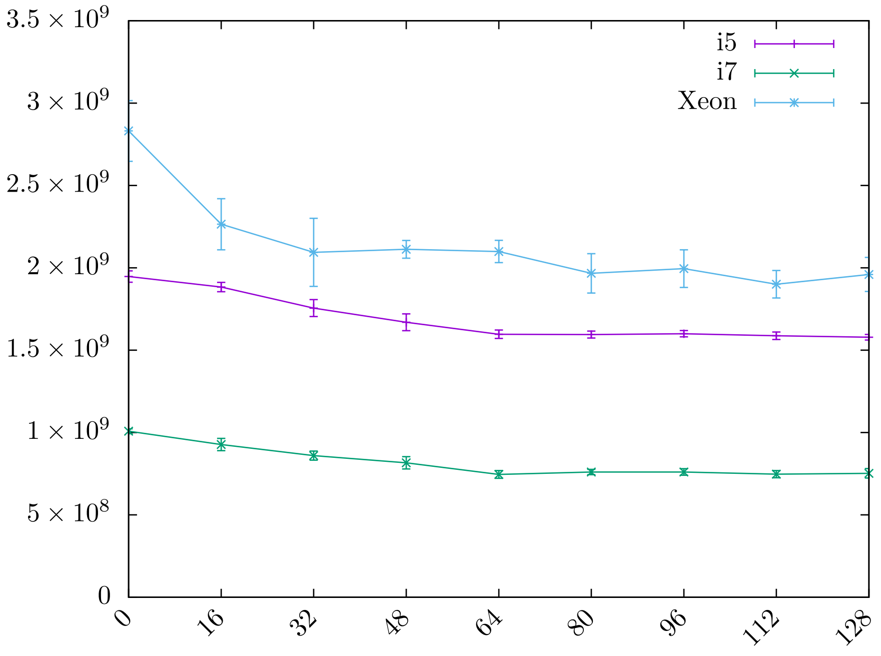


Figure 10.28: The striped-write hashmap problem on all 3 platforms. The plot shows wall-clock execution time in ns., as a function of padding in bytes.

Chapter 11

Advice for multicore programmers

{Add observations from openjdk's implementations of atomicint/atomicintarray, LongAdder, and Striped64 classes. Also from the commit message introducing @contended to openjdk, and that one blog-post suggesting to pad using inheritance. Also, padding strategies may be examined by using JOL here, if not in earlier sections}

Bibliography

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

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

[3] 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.

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

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

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

[7] 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`.

[8] 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`.

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

[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] 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`.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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