Multicore Performance Demystified

Hardware matters for java developers

Sigurt Bladt Dinesen
sidi@itu.dk

Supervisor:
Peter Sestoft

July 26, 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 70% 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: Hard drives, network interfaces, the buses that transfer data between hardware components, and address-translation hardware such as translation lookaside buffers. 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, {I'm still not sure where the cyclic reads plots will end up (here, experiment section, or "memory operations are expensive" section); get back to this} 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 gigabytes), 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 circuitry and in the operating system. The term "word" seems to simply mean "some useful number 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.

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 information 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 circuitry. For these reasons, large caches – such as the multi-kilobyte L1 caches closest to the CPU cores – are not implemented this way. 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 expertly 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, two entries whose addresses hash to the same set can be stored at a time. If a new 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 information to be copied 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 complexities of virtual memory, and the array base-pointer). 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 function, 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 mechanism 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 multi-core programming is non-trivial. As different CPU cores may 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 gives rise to the problem of false sharing.

{Continue writing: write-back caches (as described by drepper), MESI state states, store-buffers and invalidation queues (explanation of false-sharing impact)}

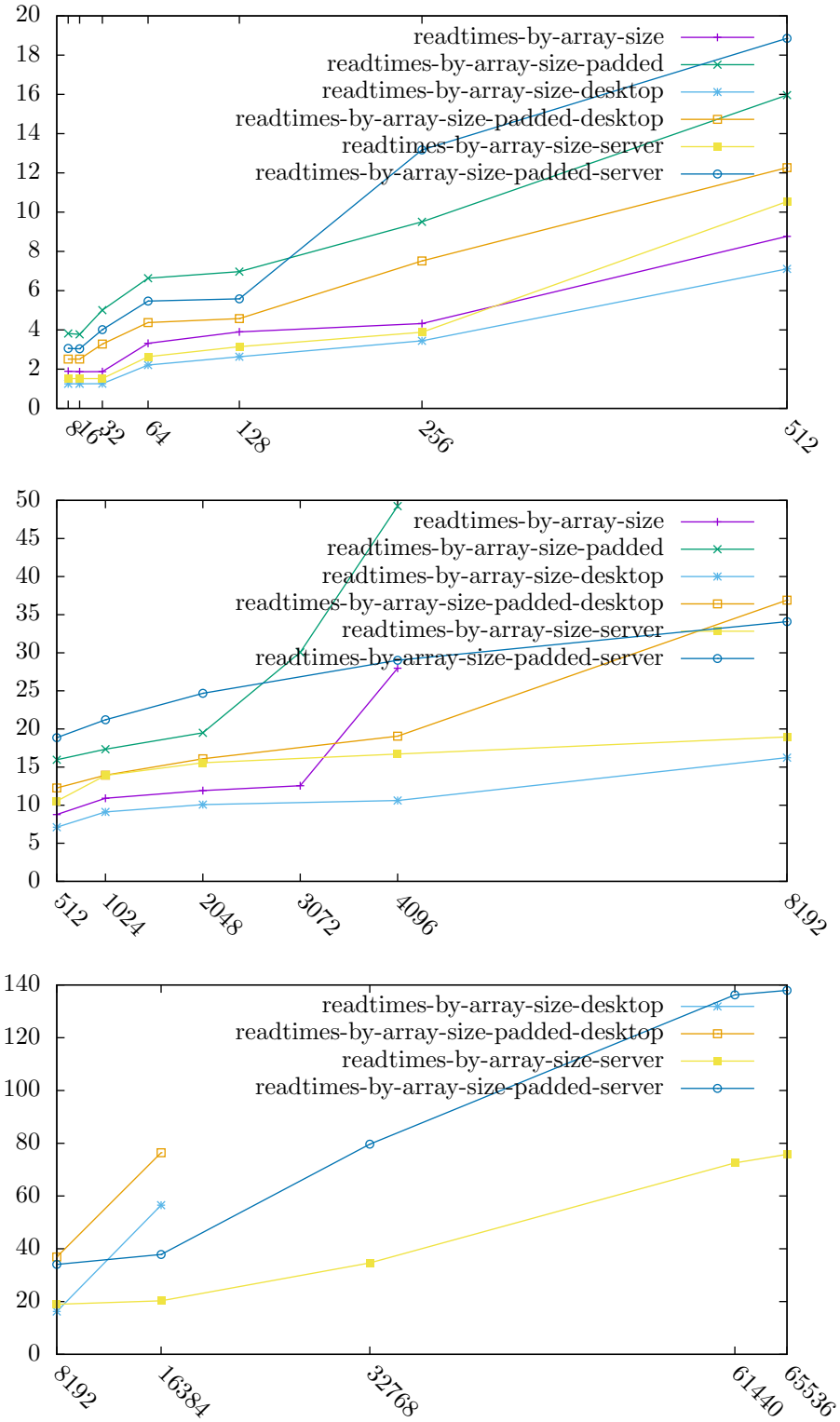


Figure 4.1: Random cyclic reads. The average read time measured over 2^{25} reads as a function of the working-set size in bytes. {I should adjust the colors so they are consistent across plots, and maybe remove the legends}

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 consumption 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

[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

sharing. However, it turns out that making simple transformations to the data layout can go a long way. In [1], adding padding around relevant variables 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 course-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 consumption for parallel programs. They 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]).

In 1993, Bolosky and Scott [10] consider 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
- is practically measurable for real programs.

It seems unlikely that any definition of false sharing will satisfy their criteria: Any definition – whether it is based on comparing a program's behaviour with that of an idealized version[3], or on analyzing sequences of memory operations

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

to determine whether unnecessary communication occurs – must either be able to distinguish the performance impacts of false sharing from those that come from communication in the memory hierarchy in general, or accept oddities like negative false sharing as optimizations to eliminate false sharing will often incur different overheads. No obvious solution for the former presents itself, and the latter 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). The is best described as hand-modifying programs to eliminate false sharing, and comparing the execution time with and without the modifications.

{Write bg for the McKenney resources, and Drepper's what every programmer...} {Write bg for the (not very hardware-specific) 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}

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

{benchmarking, limitations, and concessions (incl. background noise in bmarks, java/hw memory-layout impedance)} {parallelizability plots may go here, to describe the parameters of the experiments (or it may go in the experiments section)}

Chapter 9

The types of "time waste" in multicore programming

{"Cache-friendly" gets a new meaning}

Chapter 10

Experiments

{Comment high std dev where relevant}

10.1 Multicore Cache Performance

{contents: (parallelisability plots here?)} {I'm probably out of time, but if not, here are things to do: find problem in kmeans (and potentially qsort), implement simple dense-locking in other languages (C and C# come to mind), try a different stress-pattern for hashmaps, make cyclic-buffer CAS experiment}

To see the potential 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 multi core setting.

10.1.1 Uncontended Writes

The first example we examine illustrates the impact of false sharing by running simple, uncontended, integer-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 integers in memory.
{add code excerpt around here}

Each thread performs millions of integer-increments. The integers are uncontended; each thread has its own integer and performs no read or write operations to integers used by the other threads. Since each thread operates on its own integer, 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 integers are uncontended, we will take the difference in performance to be a result of unnecessary coherence overhead 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. The experiments are run with 4, 8, and 48 threads on the i5, i7, and xeon platforms, respectively.

Figure 10.1 and 10.2 show the average time per increment, as a function of padding.

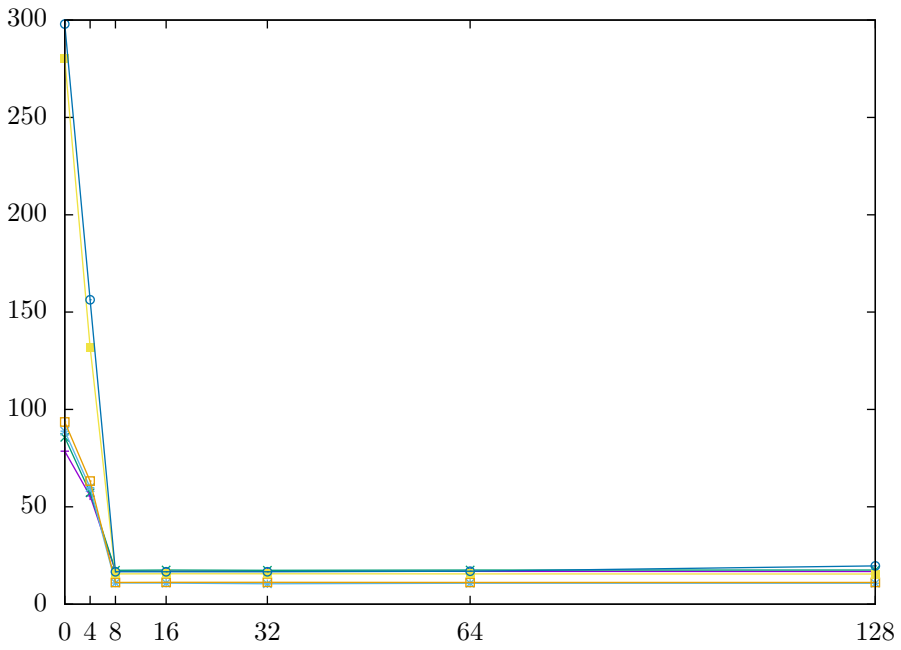


Figure 10.1: Uncontended increments on volatile integers. The plot shows nanoseconds per increment, as a function of the number of bytes used as padding between the integers.

{Add explanation of the different plots (array/local, barrier/no-barrier), include values of plots (plots are hard to read), and reflect/conclude}

10.1.2 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.
{include code snippet}

The program is similar to {reference uncontended snippet}, but in this version all threads operate on a single, shared integer. Like in the previous section, we run two versions of the experiment: One where the integer is volatile, and one where it is not. As this experiment uses a shared integer, there is no need to store it in an array. However, plots 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. It 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}

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

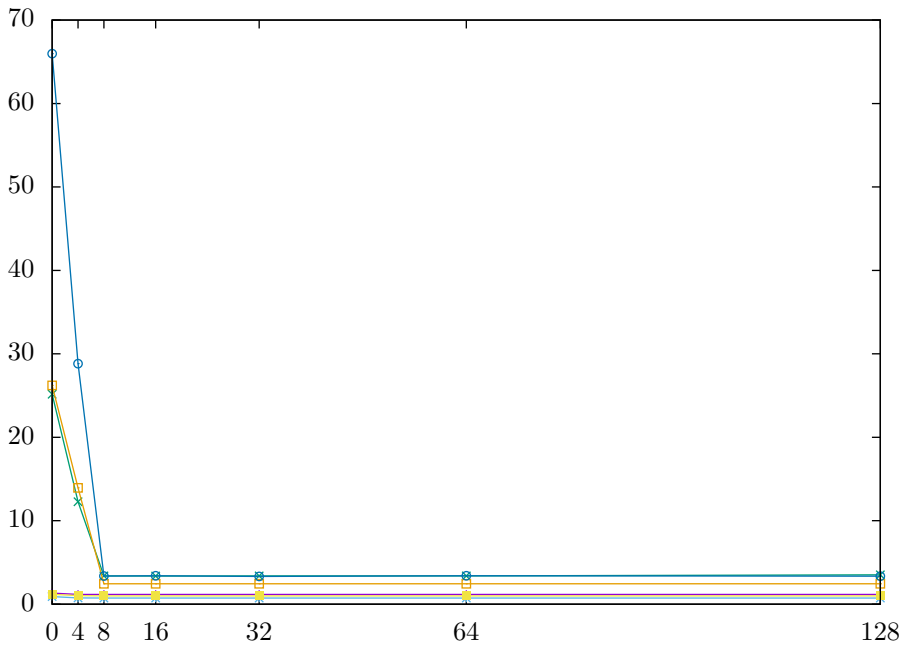


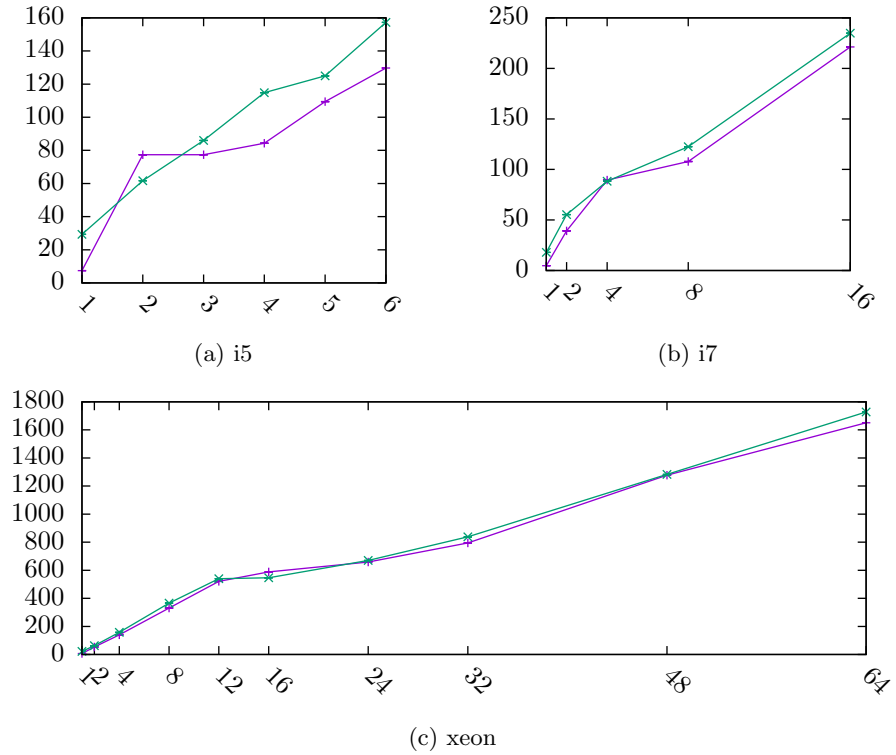
Figure 10.2: Uncontended increments on non-volatile integers. The plot shows nanoseconds per increment, as a function of the number of bytes used as padding between the integers.

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}

10.2 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 {Update if it doesn't end up 4} examples of more complicated parallel programs.

First we will look several programs that build histograms. These example programs will motivate our approach from course- to fine-grained synchronization with padding. One implementation also serves as an example of false sharing in lock-free CAS 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. We take k-means clustering and a striped hashmap

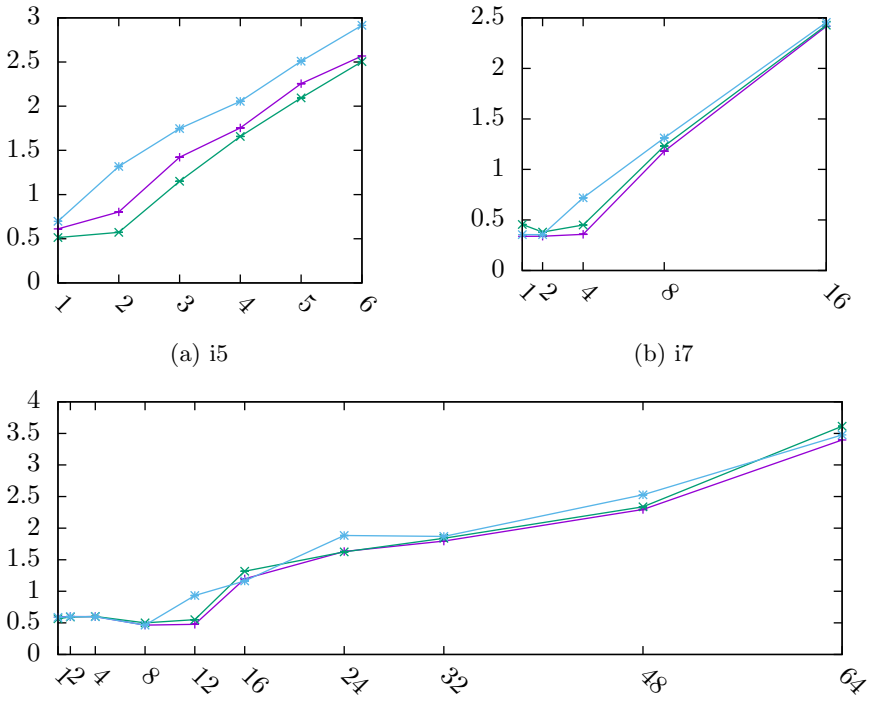


(a) i5

(b) i7

(c) xeon

Figure 10.3: Contended increments on **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.



(a) i5

(b) i7

(c) xeon

Figure 10.4: 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.

implementation as examples of locking and lock-striping applications.

10.2.1 Histogram builder

Let us consider the problem of building a histogram: Given a sequence of numbers, we wish to build a data structure that maps numbers to their frequencies in the 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: The full input sequence consists of 4 million randomly chosen integers $a \in [0, 31]$. The sequence is split between 4, 8, and 48 threads for the i5, i7, and xeon platform respectively. The more threads, the less work per thread. 32 buckets are used, one for each possible number in the input.

Course-grained locking

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 10.1: Simplified code for the threads in the course-grained locking version of the histogram problem.

The table in figure 10.5 shows the execution times of this solution to the histogram problem. At we would expect, it scales poorly with the number of cores.

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

Platform	Time (ns)	SD (%)
i5	242.6	12.9
i7	681.7	8.1
xeon	5904.5	11.7

Figure 10.5: The histogram problem using a single global lock. Time is per input-element per thread, the standard deviation (SD) is given as a percentage of the total execution time.

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 false communication 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.

```
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++;
    }
  }
}
```

The improved level of parallelism increases the potential 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 actual array elements should therefore be irrelevant, but if the objects are densely allocated, e.g. in a tight for-loop, they can still be placed back-to-back in memory.

We measure the impact of false sharing by benchmarking versions with different amounts of padding between the locks and counters.
{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.}

Figures 10.6, 10.7, and 10.8 show the total execution times for the i5, i7, and xeon platforms respectively. Each plot shows the execution time of the histogram problem, 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 effect is smallest on the i5 platform: Using 128 bytes padding between counters, and no padding between locks, yields a 10% 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.

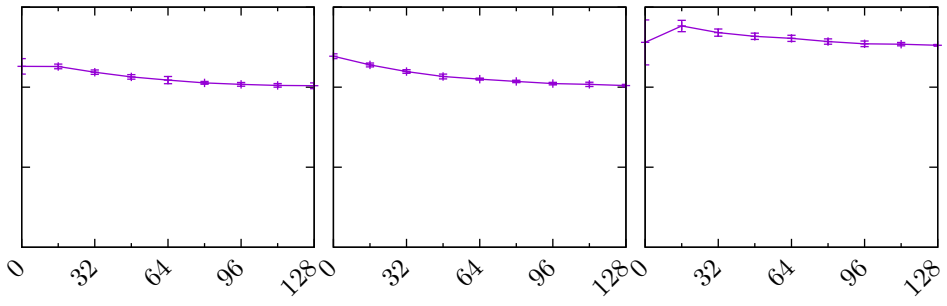
The most significant effect is seen on the xeon platform. Like on the i7 platform, using padding is beneficial for both data structures, but here the benefit from padding the locks is much more pronounced. Using 128 bytes of padding for the counters and 112 bytes for the locks yields a 71.8% improvement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

There is a noteworthy caveat: The experiment runs with 48 threads on the xeon platform, but there are only 32 locks. That 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.

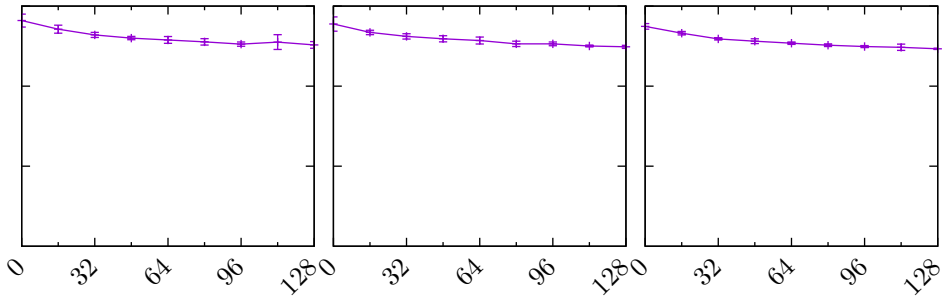
It is not surprising that the platform with the most cores suffers worst. The more threads running in parallel, the more chances to invalidate a cache line that is relevant to another core. Furthermore, the invalidation messages must be sent to all the other cores, and processors must wait for acknowledgement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

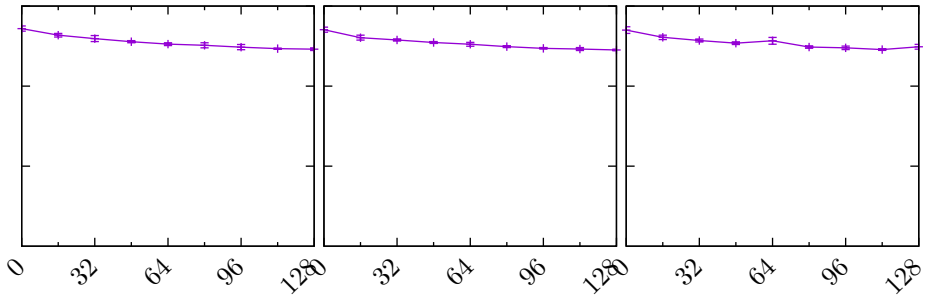
We would expect the performance to stop improving abruptly around 64



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

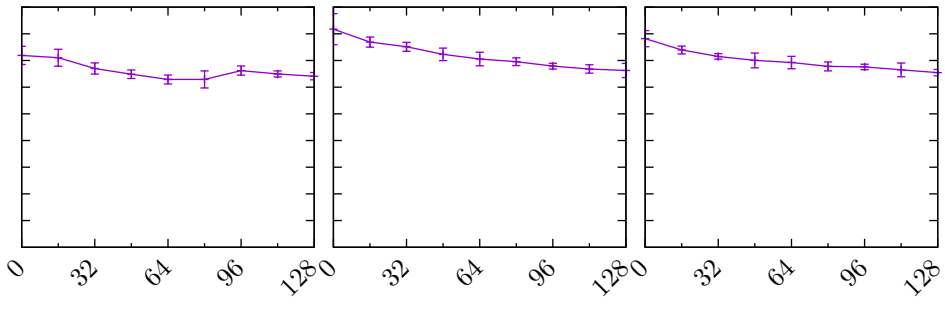


(d) 48 bytes (e) 64 bytes (f) 80 bytes

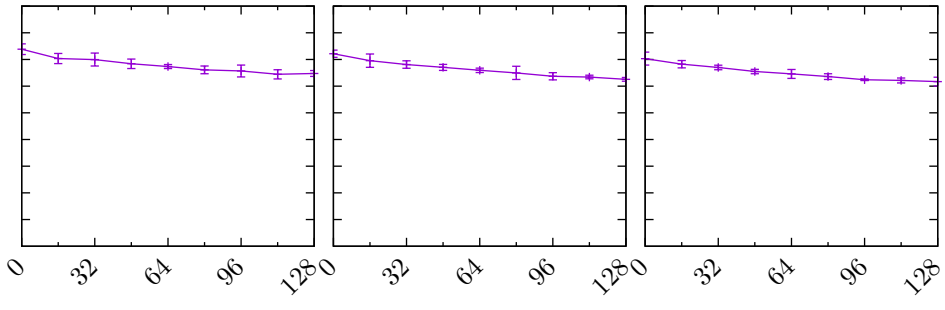


(g) 96 bytes (h) 112 bytes (i) 128 bytes

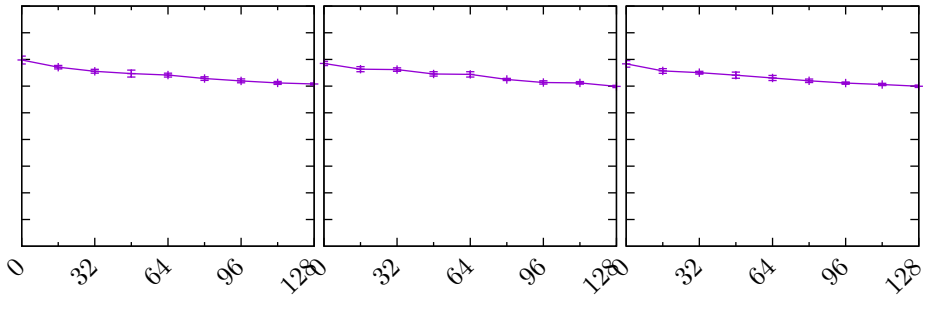
Figure 10.6: The histogram problem on the i5 platform. Here the impact of padding the locks is negative, while padding the buckets improves performance by 10.7%. The best time is 100.88 milliseconds, achieved by padding the buckets with 128 bytes, and not padding the locks. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the running 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 axis starts at 0.



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

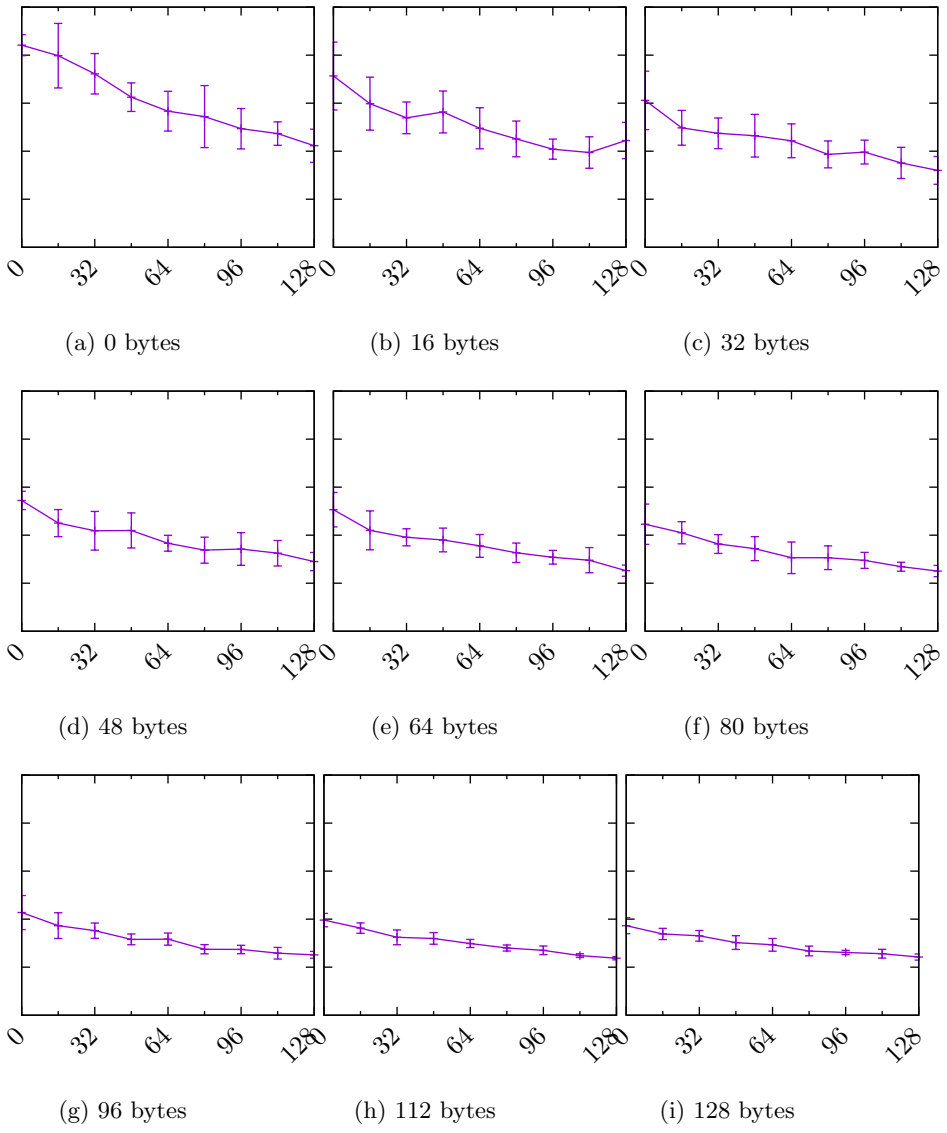


(d) 48 bytes (e) 64 bytes (f) 80 bytes



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

Figure 10.7: The 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. That is a 16.9% improvement on not using padding. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the running 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 axis starts at 0.



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

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

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

Figure 10.8: The 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. That is a 71.8% improvement on not using padding. Each plot uses a different amount of padding between the locks (specified beneath each plot). The plots show the running time of the histogram problem as a function of the amount of padding between the histogram buckets in bytes. Each tick is 100 milliseconds. The axis starts at 0.

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 `contented` annotation causes 128 bytes of padding to inserted on openjdk, under the assumption that cache lines are half that size[11]. 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.

Compare-and-swap

{Figures 10.9,10.10,10.11}

10.2.2 Divide and Conquer {avoidance? 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. With respect to synchronization, we can see this problem division as a kind of avoidance: If the sub-problems are independent, they can be solved in parallel on a multicore system without any communication - and therefore synchronization - 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.
{quicksort insert code snippet}

The program is a (somewhat naive) parallel implementation of quicksort. Anytime the algorithm divides the problem in two, the calling thread forks a task to solve the left part of the problem, executing the right part itself. Forked tasks are run in parallel at the discretion of the java `ForkJoinPool`.

As is common, and as is recommended in the popular algorithms text book by Sedgewick and Wayne [12], 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

[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.

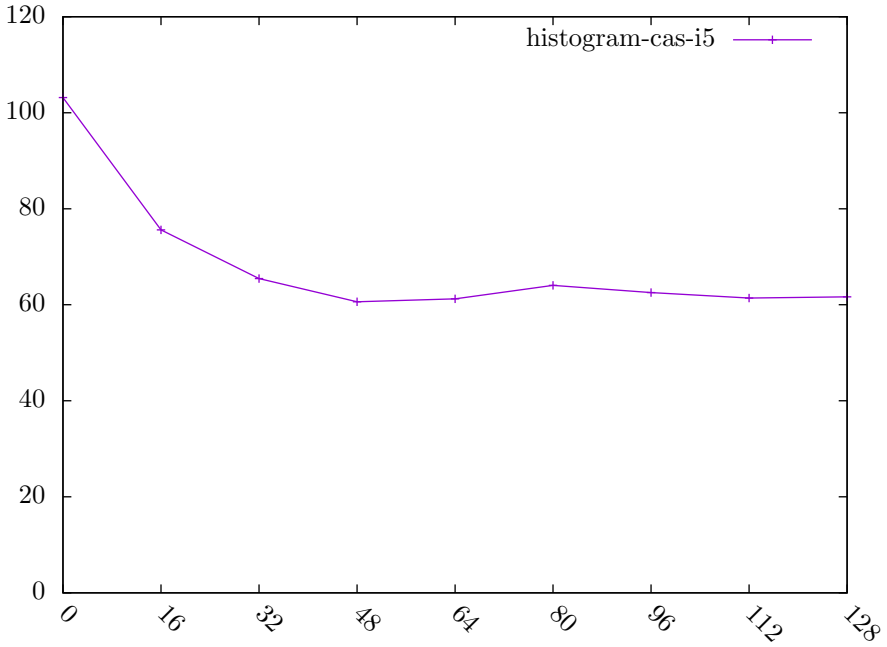


Figure 10.9: CAS i5

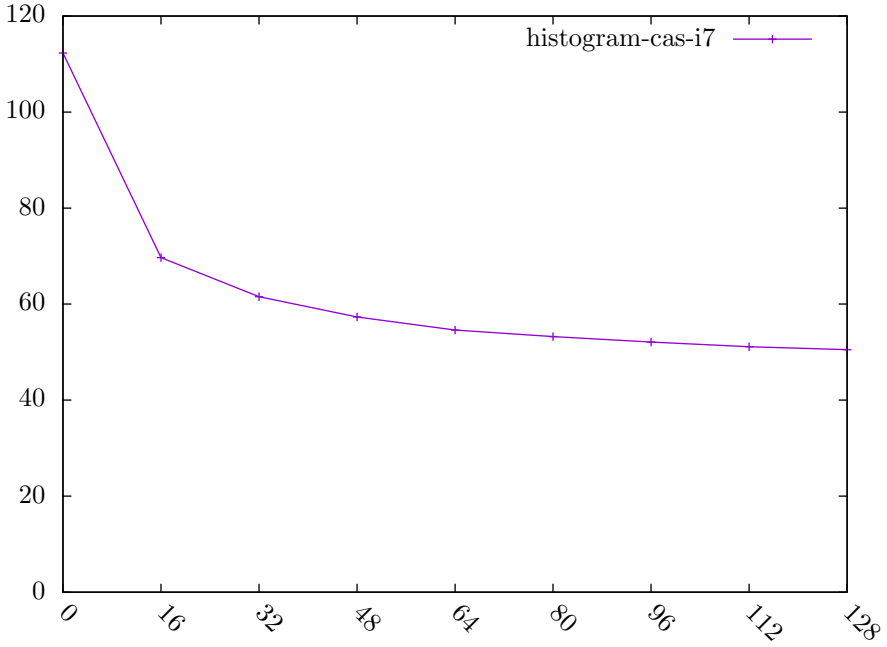


Figure 10.10: CAS i7

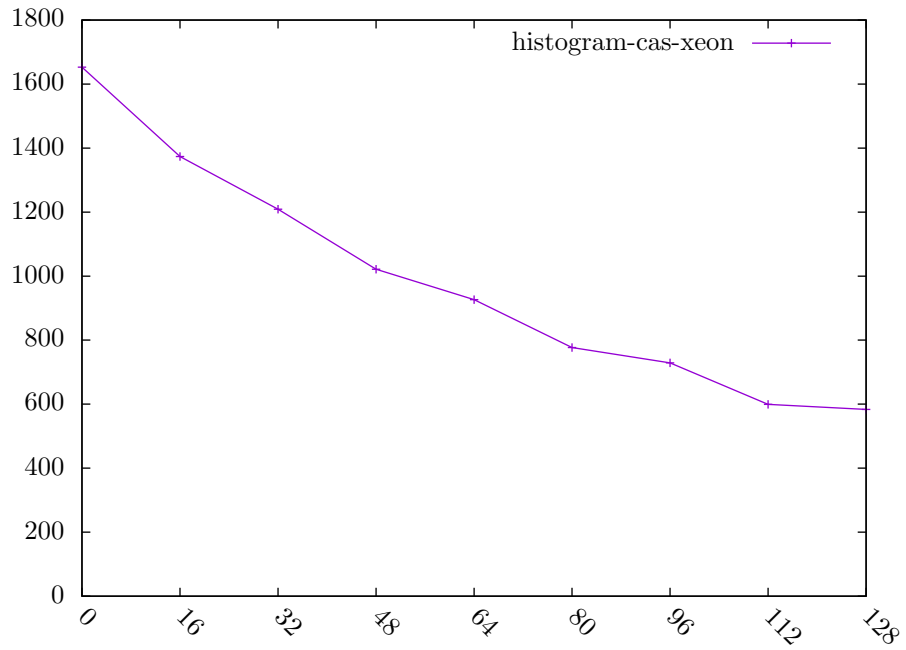


Figure 10.11: CAS xeon

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()`.
{include concrete parameters: list size(s), cutoff details}
{Analyze/conclude}

10.2.3 Locking {kmeans, striped hashmap}

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-Univeristy of Copenhagen [13, 14], As well as implementation of a concurrent hashmap datastructure used in the same course. Both of these applications are studied in [1], and can be said to have instigated this project.
{subdivide eg. with subsubsections, and complete kmeans and explanations}

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] Ulruch 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] 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.

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

[13] 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.

[14] 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.