Multicore performance demystified

Sigurt Bladt Dinesen
sidi@itu.dk

Supervisor:
Peter Sestoft

June 27, 2018

Contents

Abstract **2**

Introduction **2**

How to read this report **2**

Background **2**

Machine architecture for software designers **3**

Java and memory **3**

Memory operations are expensive **3**

Method **4**

The types of "time waste" in multicore programming **4**

Experiments **4**

Multicore Cache Performance 4

Uncontended Writes 4

Contended Writes 5

Practical Applications 6

Divide and Conquer {avoidance? Quicksort} 7

Locking {kmeans, striped hashmap} 7

Lock free{(? cas-based?)} 7

Advice for Multicore Programmers **7**

Abstract

Introduction

How to read this report

Background

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 loop transformations can affect data-locality{this might need a reference}, and thereby cache utilization, 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 they are uncontended. He subsequently shows that a lock-striping implementation of a concurrent hashmap also suffers greatly from false sharing.

While the variance 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 [2]. 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 when the cost of additional cache invalidations, and subsequent misses, outweighs the coherency overhead saved by performing fewer, larger cache reads. They also find that increasing the 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 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-

[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

grained[2] parallel programs [3] (the suite used was not the same as in [2]). 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. [4] analyzed the effects of false (as well as true) sharing 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, we are not aware that this is case for any particular compiler. Their work shows that focusing on data sharing is particularly important when using optimizing compilers. This is not because optimizing compilers make sharing worse, but because they are good at eliminating memory accesses to processor-private data e.g. by keeping data in CPU registers. Hence, when using an optimizing compiler, memory access to shared data often dominates IO utilization for parallel programs. They provide a set of optimizations requiring detailed profiling information about the program to be optimized, and a set 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 vs. 4-256 bytes in [3]).

In 1993, Bolosky and Scott [5]

{Write bg for torrellas and Bolosky&Scott}

{Write bg for the resources used at the outset of the project}

Machine architecture for software designers

{(focus on memory hierarchy and coherency)} {readtimes plots go here.} {Caches, MESI, store-buffers and invalidation queues (explanation and proof of false-sharing impact)}

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}

Memory operations are expensive

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

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

Method

{benchmarking, limitations, and concessions (incl. background noise in bmarks, java/hw memory-layout impedance)}

The types of "time waste" in multicore programming

{"Cache-friendly" gets a new meaning}

Experiments

{Note high std dev where relevant}

Multicore Cache Performance

{contents: (parrallelisability?)}

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 integer-increment operations in a multi core setting.

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 2 and 1 show the average time per increment.

{Include concrete parameters: work is not divided between threads, but is additive, lap/desk/serv performs x,y,z ops..}

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

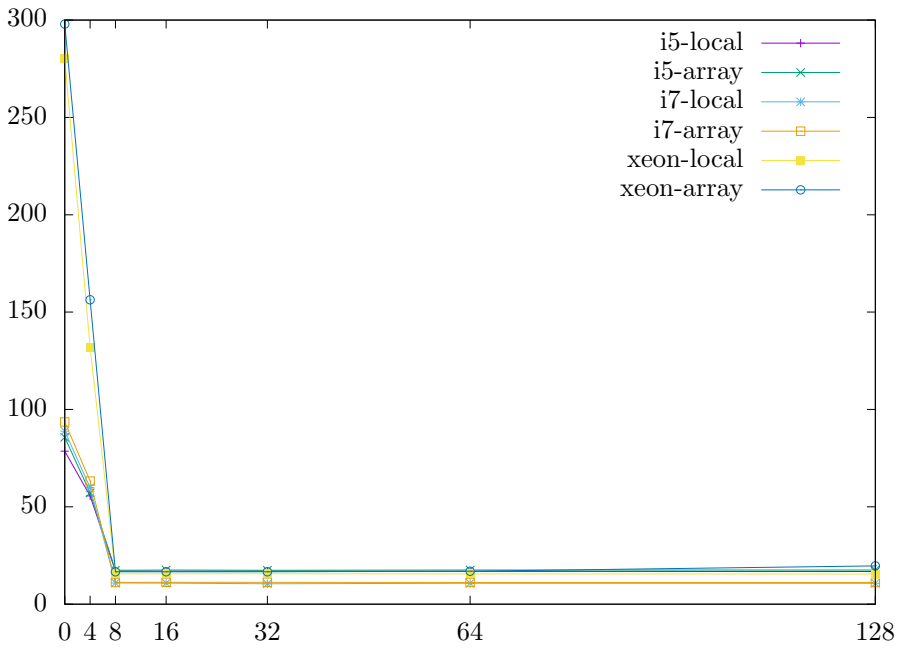


Figure 1: Uncontended increments on volatile integers. The plot shows nanoseconds per increment, as a function of the number of bytes used as padding between the integers.

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 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.

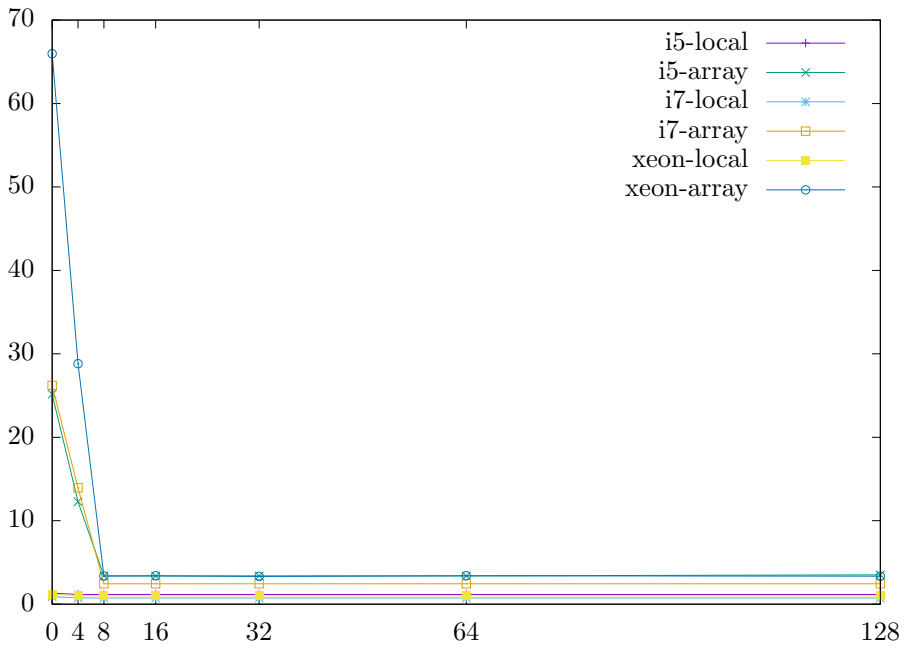
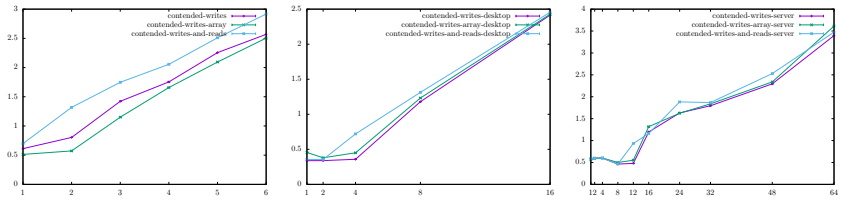


Figure 2: Uncontended increments on non-volatile integers. The plot shows nanoseconds per increment, as a function of the number of bytes used as padding between the integers.

{include table of numbers (they are 833.4±2.82 (desktop), 1940.5±58.12 (server), and 1256.7±13.59 (laptop) ns)})



(a) (1a) (b) (1a) (c) (1a)

{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}

Practical Applications

We rarely make software that simply increments counters, it is therefore useful to see the effects of false sharing in more complicated applications as well.

In this section we examine false sharing in 4 {Update if it doesn't end up 4} parallel programs. We 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 implementation as examples of locking applications. Finally, we {more problems go here when found}

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.

{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 [6], 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 array. 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}

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 [7, 8], 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 subsubsuectiont, and complete kmeans and explanations}

Lock free{(? cas-based?)}

{this needs to be done}

Advice for Multicore Programmers

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.