Matan Lachmish, Asaf Rokach

Prop. Yehuda Afek

Distributed Computation

1 August 2016

Bounding a Total Store Ordering (TSO) Memory

Matan Lachmish mlachmish@gmail.com

Asaf Rokach style007@gmail.com

1 Abstract

This work address the issue of bounding a total store ordering (TSO) memory. The issue was introduced in Afek's and Morrison's paper "Fence-Free Work Stealing on Bounded TSO Processors" [1]. Their paper introduce a major optimisation to the "Work Stealing" method, demonstrating work stealing algorithms in which a worker does not issue a memory fence for microarchitectures with a bounded total store ordering (TSO) memory model.

Their optimisation relays heavily on the fact that the memory model have a bounded total store order, which means the number of stores a load can be reordered with is bounded. This is a novel restriction of TSO – capturing mainstream x86 and SPARC TSO processors.

We intend to **prove the fact that the bound do exist** and even try to find the exact boundary.

This paper covers three methods to measure the store buffer size, as some have better results on real systems.

We Implemented the three methods mentioned in this paper and test them on several architecture, gathering some very insightful results.

You can find the implementation in www.github.com/mlachmish/TSOBounder

2 Introduction

Afek's and Morrison's paper "Fence-Free Work Stealing on Bounded TSO Processors" approach the field of task-based parallel programming model. The implementations of task-based parallelism dominantly employ work stealing for dynamic load balancing of the executed tasks.

In work stealing, each worker thread has a queue of tasks from which it continuously removes the next task to execute. While executing a task the worker might create and add new tasks to its queue. If a worker's queue empties, the worker becomes a thief and tries to steal a task from another worker.

Today's work stealing synchronization protocols are based on the flag principle. The worker publishes the task it is about to take, and then checks whether a thief intends to steal the same task. If not, the worker can safely take the task because any future thief will observe its publication and avoid stealing the task. But for this reasoning to hold, the worker must issue a costly memory fence instruction to prevent the checking load from being reordered before the publishing store.

Afek's and Morrison's idea was to challenge Attiya et al.'s "laws of order" [2], as in their words:

> "laws of order rely on certain assumptions and may not hold when these underlying assumptions are invalidated"

Their paper demonstrates that linearizable fence-free work stealing is possible on mainstream multicore architectures with a total store ordering (TSO) memory model, such as x86 and SPARC.

Their paper only assumed that this bound can be found, our intention is to design and implement an algorithm that will determine the bound of the store buffer.

The algorithm we design should work on a real working machine, therefor it should be resilient to a “real life” unexpected behaviour of noise and context switches.

3 Measuring Methods

All the methods described bellow can be found at

www.github.com/mlachmish/TSOBounder under ‘bounder.py’ module.

3.1 Chalenges

A key feature that is common to all of the bellow algorithms is that no matter what is the measurement we are taking, we are always averaging it over a big number of tests (~100K). The reason for this is because the computers we tested on are real working computers with of the self operation system. We could not interfere with the way the system schedule cpu time for our thread. So instead of that we performed each measurement multiple times in order to average out every unexpected context switch or other noise from the system.

3.2 Coarse Grain

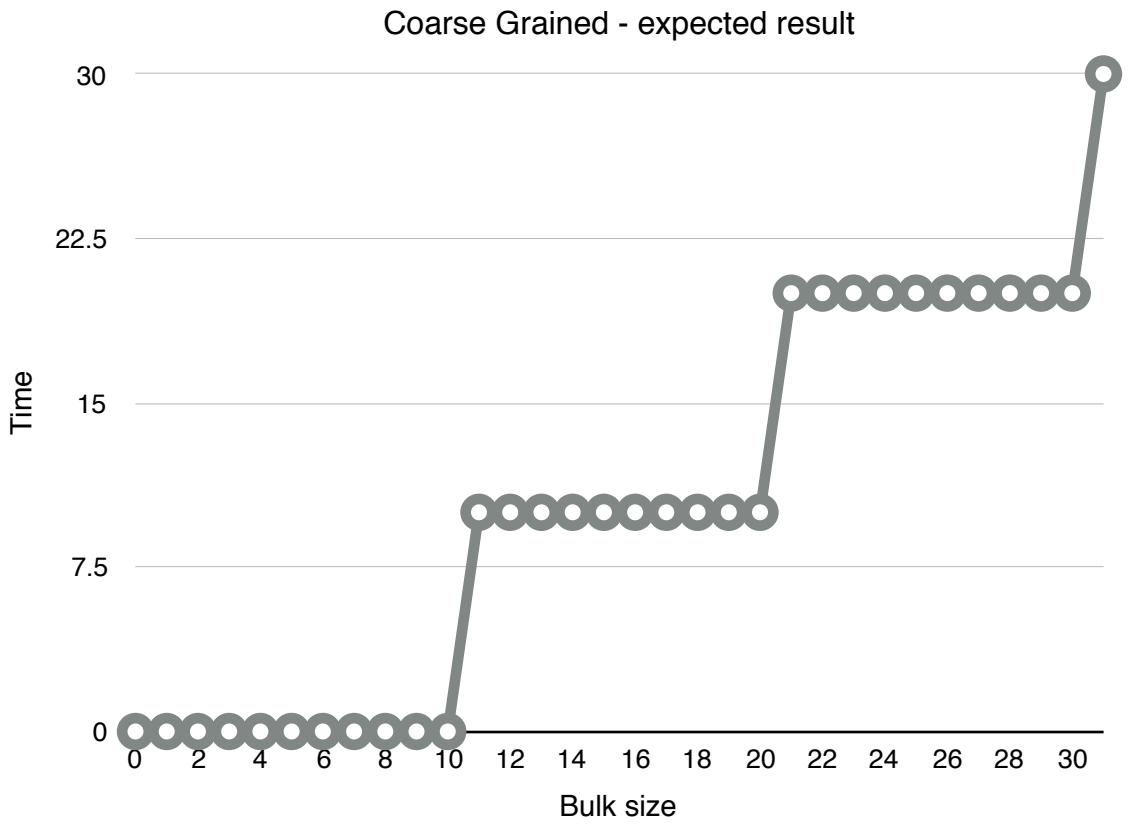
The first algorithm we implemented was the coarse grain. This algorithm was introduced briefly in Afek’s and Morrison’s paper.

Algorithm 1 Coarse Grain

```
1: procedure CoarseGrain                    ▷ Measure write time of bulk writes
2:     resultArray = []
3:     for bulkSize = 1, ..., kNumberOfWritesPerTests do
4:         startTime = currentTime()
5:         for testNumber = 1, ..., kNumberOfTests do
6:             writeBytes(bulkSize)
7:         endTime = currentTime()
8:         resultArray.append([bulkSize, (endTime-startTime)/kNumberOfTests])
9:     return resultArray
```

This algorithm measure how long does it takes to write different sizes of bulks. The Idea here is that as long as the bulk size is less then the store buffer size the mea-

sured time should be almost the same. Once we reach the buffer size (+1), the measured time should be significantly high - a non linear step. The result should look like a step graph, where each step width is the store buffer size.



The result we got from this algorithm were very noisy and not satisfying, therefor we came up with the “Fine Grain” algorithm(3.3).

3.3 Fine Grain

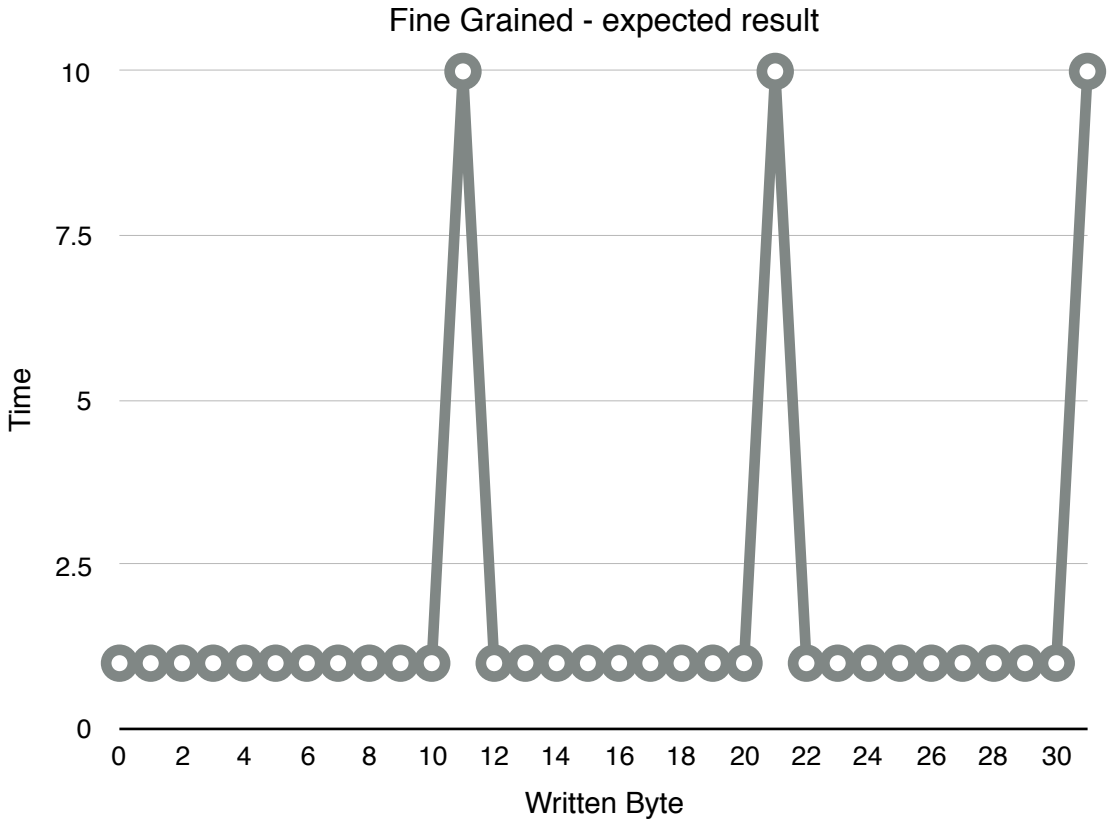
The fine grain algorithm uses the measurement idea suggested in the coarse grained algorithm but with an approach that gives noise less affect on the measurement.

Algorithm 2 Fine Grain

```
1: procedure FineGrain                    ▷ Measure write time of single writes
2:     resultArray = []
3:     for testNumber = 1, ..., kNumberOfTests do
4:         startTime = currentTime()
5:         for byteNumber = 1, ..., kNumberOfWritesPerTests do
6:             writeBytes(1)
7:             endTime = currentTime()
8:             resultArray[byteNumber] = (resultArray[byteNumber] *
   (testNumber - 1) + (endTime - startTime)) * testNumber▷ Average
   time to write specific byte
9:     return resultArray
```

This algorithm measure how much time it takes to write each byte a long write sequence.

As in the coarse grained algorithm, we repeat every test multiple times. That means that after all the test we have the mean time of writing the first byte, the mean time of writing the second byte, etc… That means that every store instruction that reach the store buffer should take a very short time, while the store instruction that will cause the buffer to flush will cost a heavy time penalty. The result should look like a spikes graph where every K spike (K is the buffer size) is noticeably longer then the prior K spikes.



The result we got was a lot better then the Coarse grained results but still we experienced a lot of noise in the result graph. This is when we figured out we might need to first flood the buffer(3.4).

3.4 Buffer Flood

In attempt to figure out why do our measurements contain so much noise, we figured that the start conditions for each test within the algorithm are not the same. For instants lets say that the buffer size is 10B and we repeat our fine grain test 100 times. Every test might (and likely is) start when the buffer current capacity is different, therefore the measurement we are taking might be in an offset. For example, if at the beginning of the test the buffer already contains 5 bytes we will get a measurement that is mis-aligned by 5 bytes to a test that starts with an empty buffer.

This is why we designed the “Buffer Flood” routine that should be triggered before every test. The “Buffer Flood” floods the store buffer until a point we believe that the buf-

fer is empty (just after a flush), this way we are more likely to achieve the same start conditions to all tests.

The way that the “Buffer Flood” works is that it writes byte after byte, each time taking a measurement. When a single write takes more then a certain **threshold** more then the average of all writes until that time, we assume that we had reach the desired flush.

An important property is the threshold, after testing on several machines and analysing the flush behaviour we found that the 1.8 factor behave like a good threshold. But of course this is just an empirical result and can be vary from one system to another.

Algorithm 3 Buffer Flood

```
1:  procedure BufferFlood ▷ Flood the store buffer until we reach a flush
2:      meanWriteTime = 0.0
3:      for testNumber = 1, ..., kNumberOfTests do
4:          for byteNumber = 1, ..., kNumberOfWritesPerTests do
5:              startTime = currentTime()
6:              writeBytes(1)
7:              endTime = currentTime()
8:              currentWriteTime = endTime - startTime
9:              if currentWriteTime >= meanWriteTime * 1.8 and
    byteNumber > 1
10:                 return True
11:             else
12:                 meanWriteTime = ((meanWriteTime * (byteNumber - 1))
    + currentWriteTime) / byteNumber
13:     return False
```

4 Evaluation

We have implemented all of our algorithms in Python 3,

You can find the implementation in www.github.com/mlachmish/TSOBounder

We then evaluated them on two different machines:

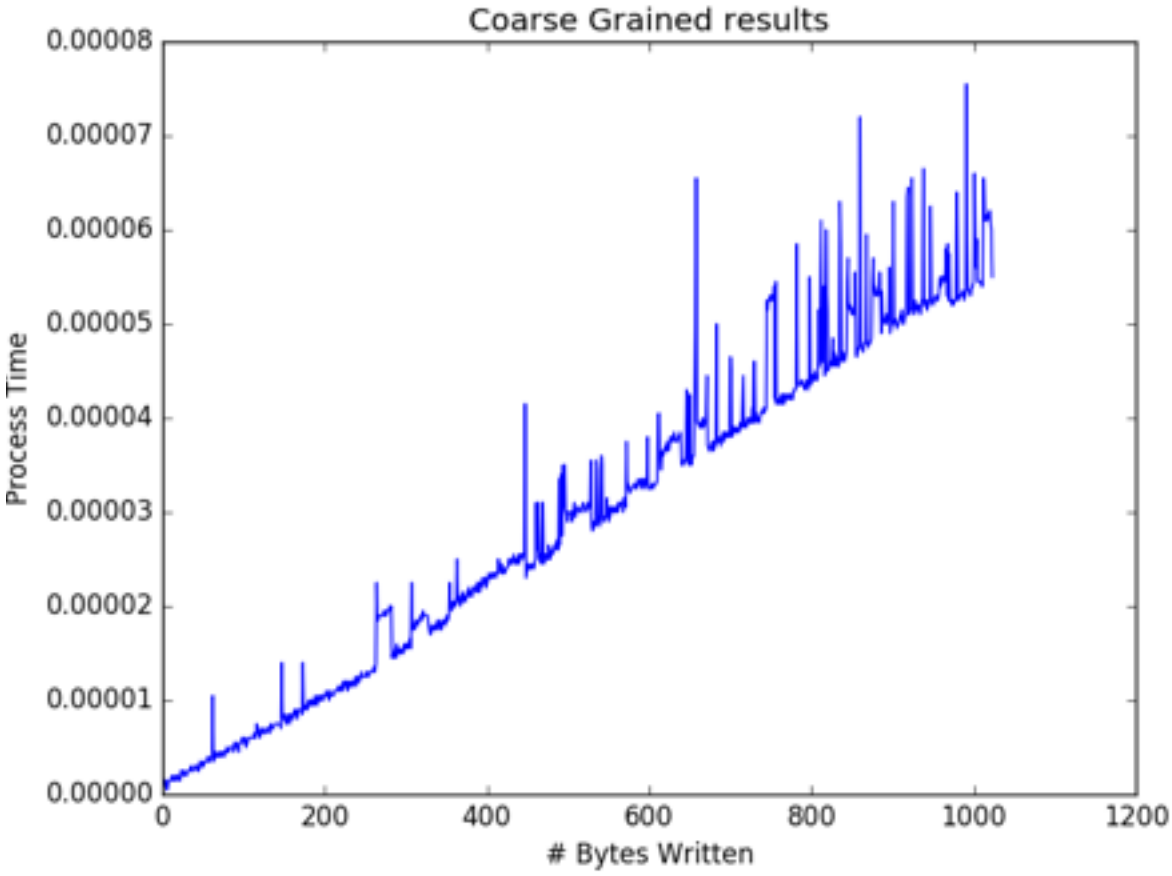
- MacBook Pro (Retina, 15-inch, Mid 2015) - i7 (4980HQ)
- Dell Vostro v131 (13-inch, Mid 2013) - i5 (2430M)

The results described below are from the most mature permutation of each algorithm, i.e every test start with a “Buffer Flood” and with the best parameters for “Buffer Flood Threshold”, number of tests, etc…

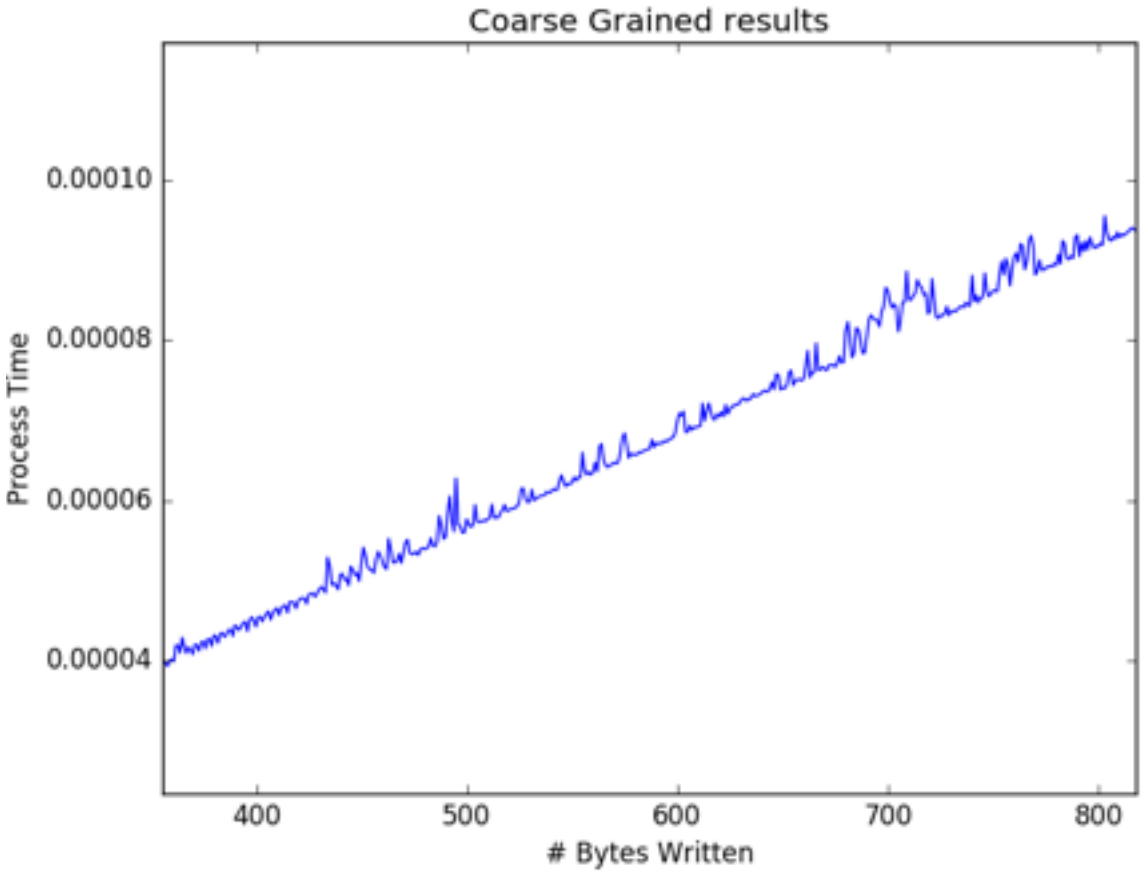
4.1 Coarse Grain

As described in section 3.2, the coarse grain results were generally not productive. We could not evaluate the size of the store buffer in a definitive way.

i7 (4980HQ):



i5 (2430M):

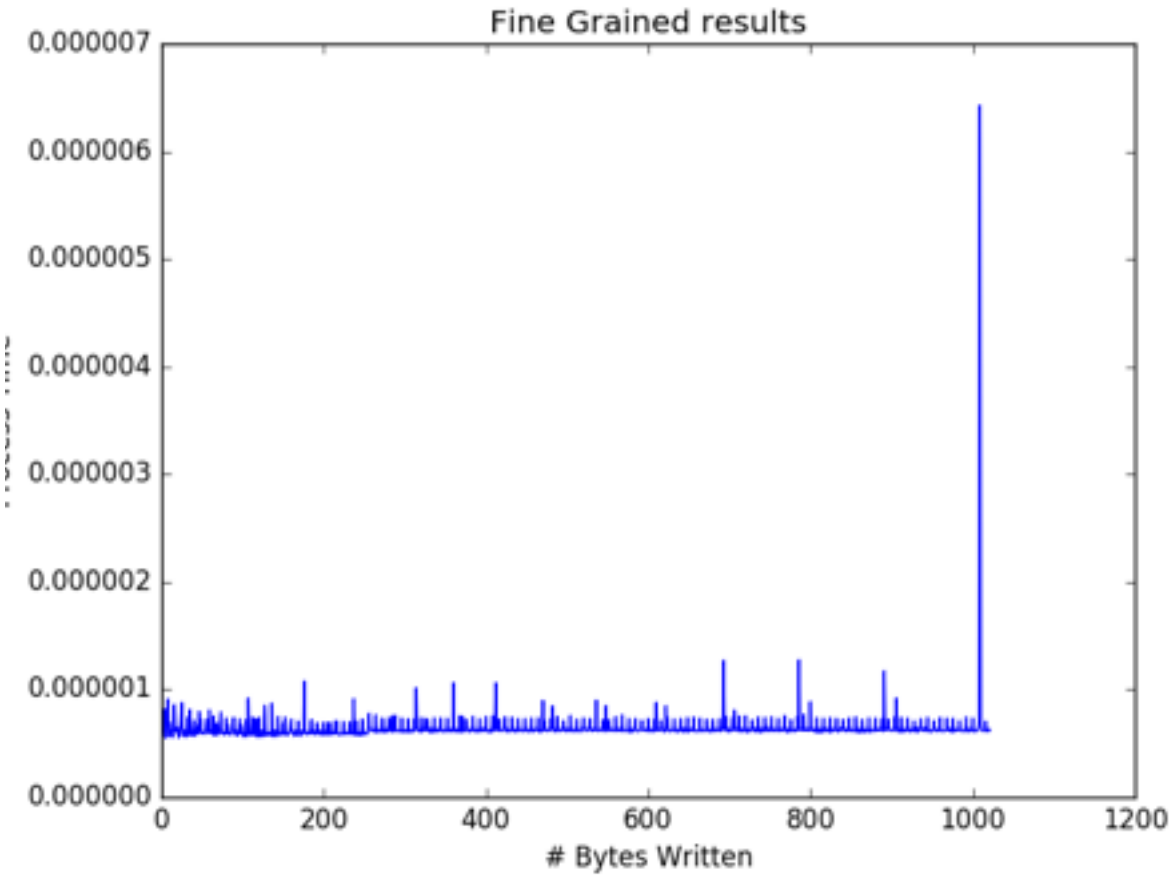

Coarse Grained results
Process Time
0.00010
0.00008
0.00006
0.00004
400
500
600
700
800
Bytes Written

4.2 Fine Grain

The result graph display nice repetitive pattern of spikes just as we predict in section 3.3. From the result we could easily identify what is the suspect of being the store buffer bound.

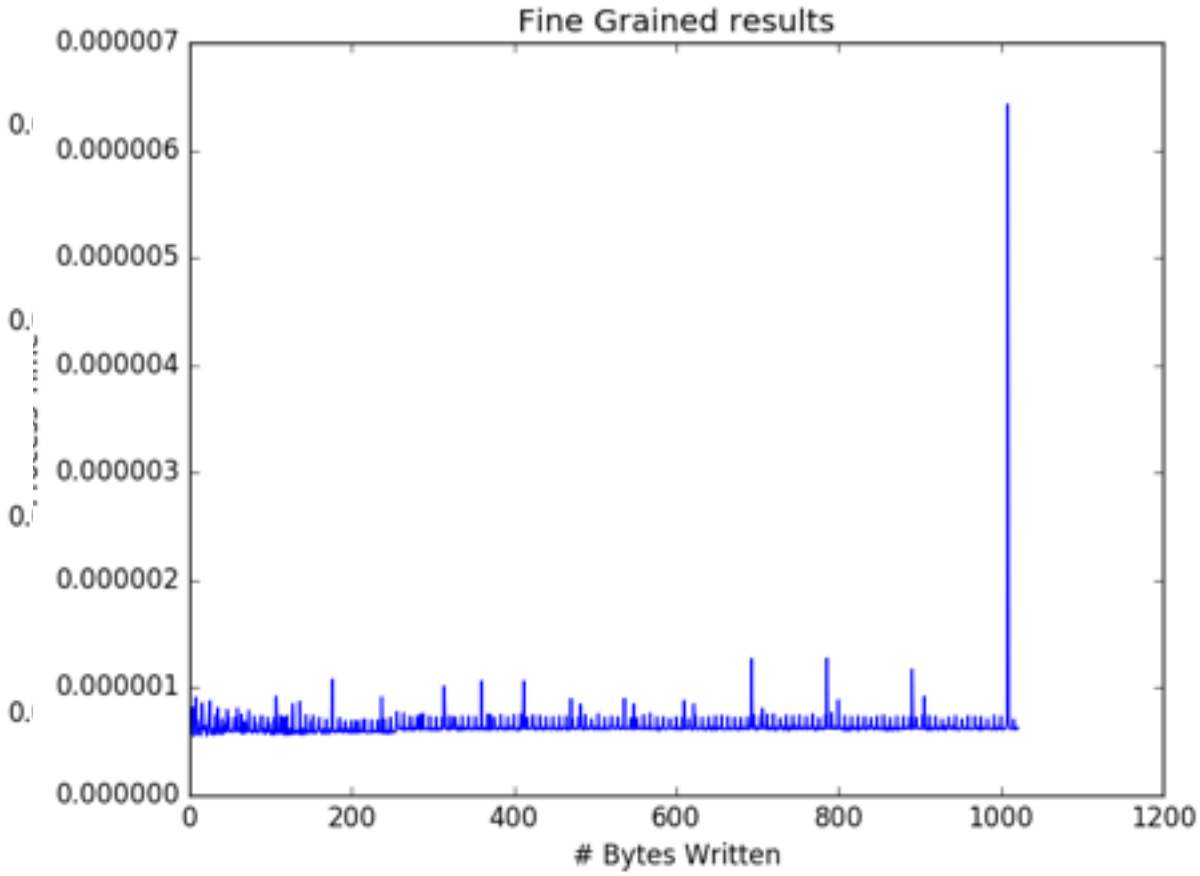
i7 (4980HQ):

The distance between each to big spikes is 64 Bytes.



i5 (2430M):

The distance between each to big spikes is 82 Bytes



5 Conclusion and future work

In conclusion, the good news is that the assumption Afik and Adam took turns out to be pretty accurate - **on a modern Intel architecture the store buffer is bounded**.

In practice it is not simple to measure and estimate the store buffer size due to a lot of noise that exist in a ruling system.

It turns out that the “Fine Grain” method was far more productive then the “Coarse Grained” method we had started with, it does make scene because the fine method leaves a very little room for noise to interfere.

The “Buffer Flood” theory proved to sharpen the result we got from both the fine and the coarse methods.

As for future work we believe that all the “Manual” tasks we did, like understanding from the graph what is the buffer size, can be achieved by code. That way a parallel job stealing algorithm can determine in a dynamic way what is the amount of instructions it needs to wait in order to avoid the uses of flags.

More then that we can larvae machine learning techniques in order to find the exact spot of the store buffer bound.

We can even try and find ways to make the Coarse Grain result “fit” the steps graph we predicted by constraining the graph to the closest step-like graph.

6 References

[1] Yehuda Afek, Adam Morrison, Fence-Free Work Stealing on Bounded TSO Processors, Tel Aviv University, 2014

[2] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pro- gramming Languages, POPL ’11, pages 487–498, New York, NY, USA, 2011. ACM.