Lab Report #3

CASSARD Sebastien 19960526-T313

November 11, 2018

1 Raw results:

1.1 Mutex lock

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	1,62s	6 045 620 819	1 721 553 805	199 762
2	5,35s	13 222 059 660	3 564 390 543	56 876 518
4	4,94s	13 769 662 620	3 727 566 446	85 726 698

1.2 Basic CAS lock

This lock is implemented using a Compare and Swap atomic instruction. The latter is used when acquiring the lock, but not when releasing the lock (in our program, when the unlock release function is called, the lock value is necessarily 1).

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,69s	1 330 547 211	442 945 574	370 153
2	3,21s	2 166 260 302	652 544 211	74 814 537
4	6,20s	4 497 775 771	1 245 982 490	136 007 213

Compared to the previous lock, the basic CAS lock seems more efficient for a small number of threads. For a large number of threads, the performance of the CAS lock is worse, especially because the threads are constantly trying to acquire the lock until they get it.

1.3 Optimized CAS lock with yield

As we have seen before, the weak point of the basic CAS lock is that threads constantly try to access the lock until they have it. Here we will consider an improved version of this lock, in which if a thread fails to acquire a lock, it yields the processor. This version offers better performance than the basic CAS lock for a large number of threads.

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,67s	1 324 317 917	441 150 548	106 993
2	0,81s	2 358 698 096	779 336 794	6 130 807
4	0,86s	3 116 518 079	1 003 780 514	5 105 680

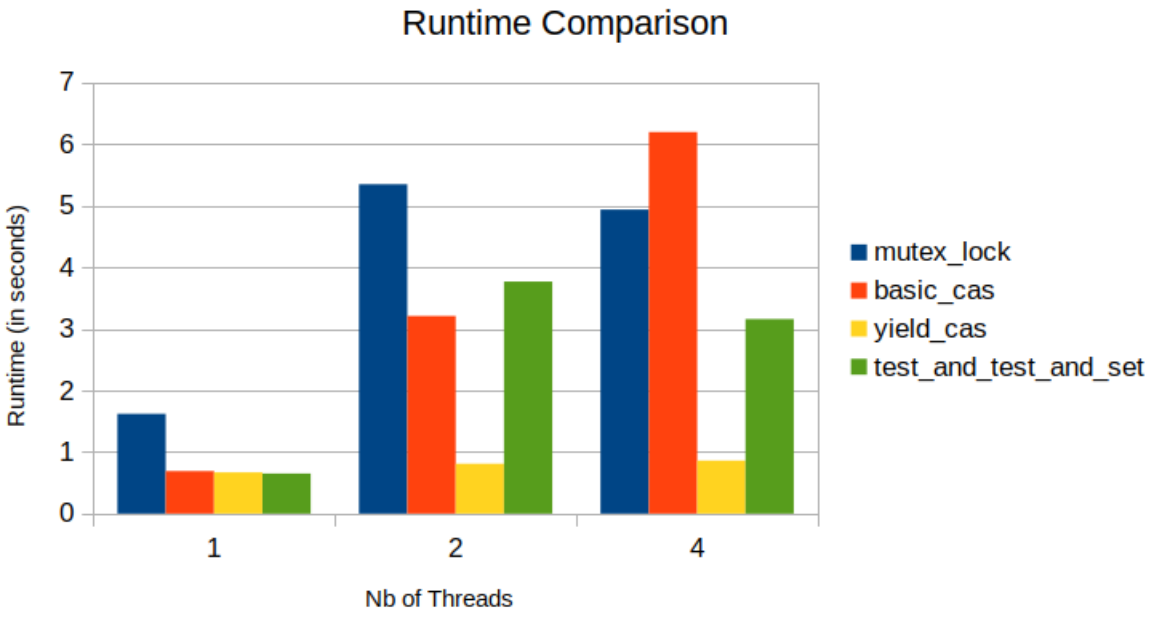
1.4 Test and test and set

This last implementation is a test and test and set lock. Its objective is to reduce the number of atomic operations compared to previous implementations. Its procedure is simple, we try to acquire the lock with a CAS, if we do not succeed we check the value of the lock until it changes, then we retry to acquire the lock with a CAS.

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,65s	1 324 624 108	441 242 454	121 550
2	3,77s	4 981 871 508	1 365 046 754	116 194 182
4	3,16s	9 814 681 971	2 568 246 674	84 994 492

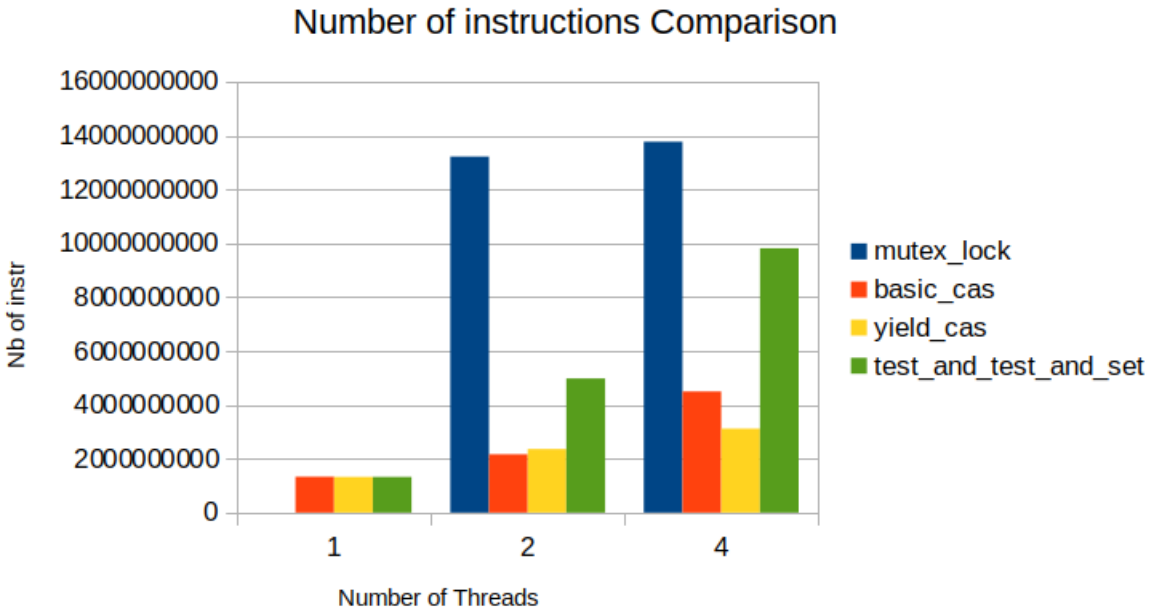
2 Comparison

2.1 Runtimes



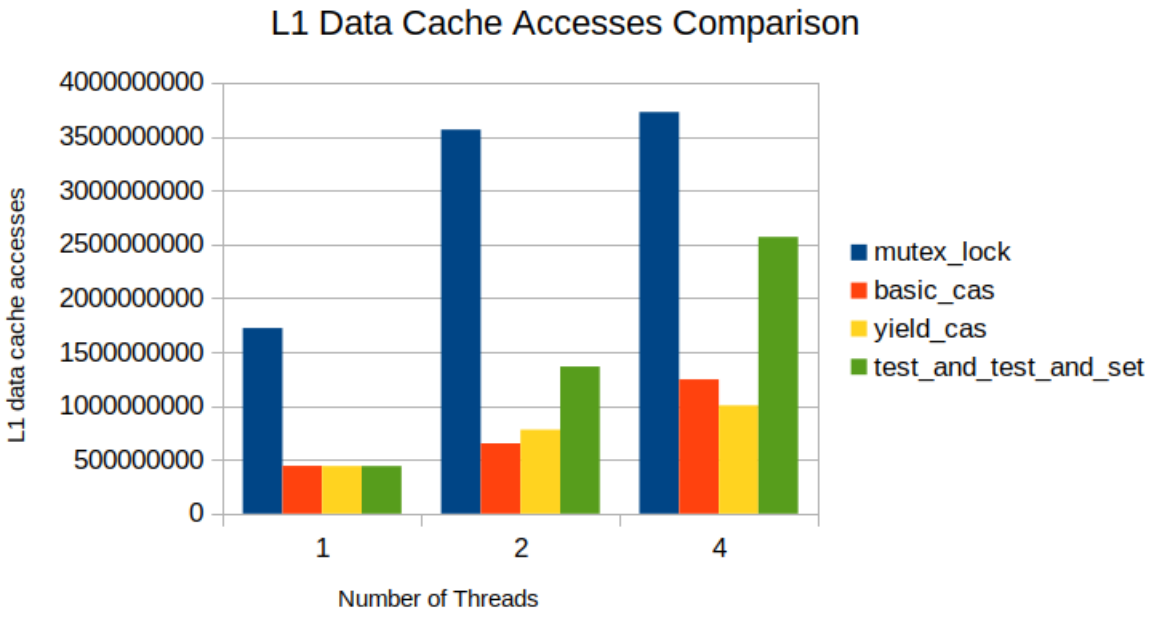
Concerning the runtime, it seems that the yield CAS outperforms all other implementations. We can see that the basic CAS gives weak results on a high number of threads, while the T&T&S seems to have better performance for a high number of threads.

2.2 Number of instructions



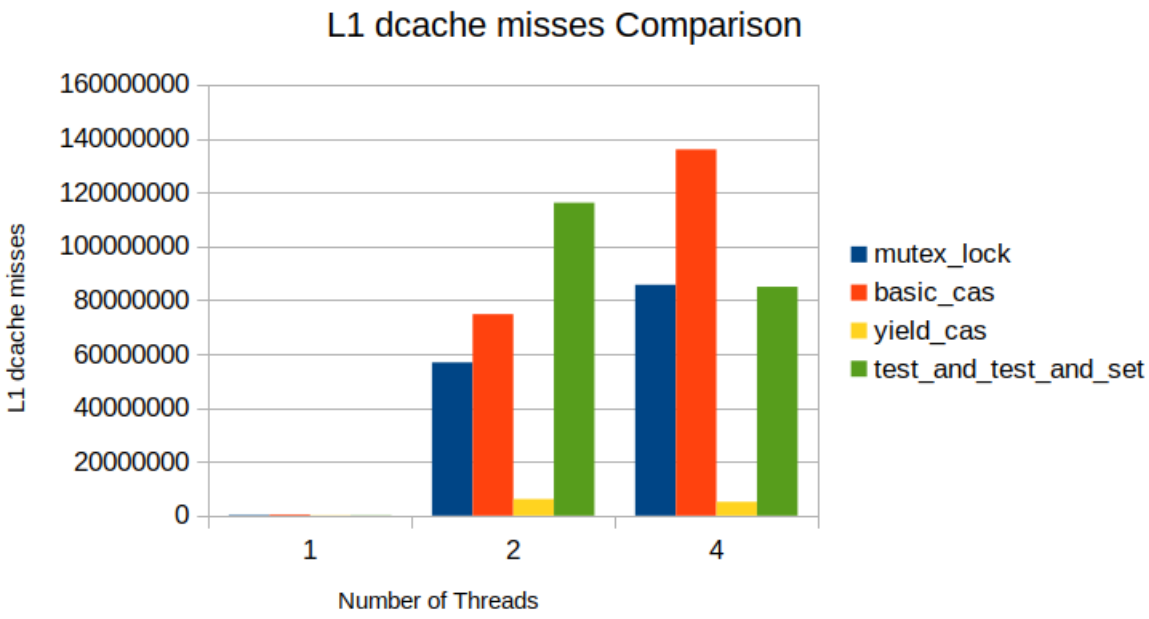
The number of instructions of the two versions of the CAS lock are quite similar for 1 and 2 threads, but for 4 threads, it is the yield CAS that has the lowest number of instructions.

2.3 L1 data cache accesses



In terms of data cache accesses, yield CAS seems more interesting than the others.

2.4 L1 dcache misses



According to the results, the yield CAS seems to outperform all the others in terms of cache misses. However, these values are so surprising that it could most likely be a measurement error. Indeed, given the previous measures, one would rather expect similar results between the basic CAS and the yield CAS. As for the T&T&S, we always notice that its performance improves with the increase in the number of threads.

2.5 Conclusions

After studying the results of the different implemented locks, we notice that for a small number of threads (in this case 1 or 2) the basic CAS lock and yield CAS lock have the best results. In return, for a larger number of threads (here 4), the basic CAS lock is not really viable, and we will rather choose the yield CAS lock which seems to present very good results. As for the T&T&S lock, we would have expected it to perform better for a large number of threads. Here the T&T&S lock results do not give great results. Maybe for an even higher number of locks we would see much better performance compared to other locks.