SimuLock - Simulator for lock based synchronization primitives on many-core processors

Ajit Singh
IT Dept, Indian Inst. of Info. Tech
Allahabad, India
rsi2016004@iiita.ac.in

Dr. Pavan Chakraborty
IT Dept, Indian Inst. of Info. Tech
Allahabad, India
pavan@iiita.ac.in

***Abstract*—Simulock, is a queueing network based simulator for lock based synchronization algorithms, that simulates execution traces of benchmark application generated through instrumentation of applications. Its a graphical tool showing interaction of different processor components(cores, caches, NOC) while executing multi-threaded SMP application, synchronized using different locking approaches. A detailed report generated after analyzing an array of locking primitives with multiple scheduling approaches(FIFO, LIFO, SRPT, RANDOM ...) is provided to the user, allowing him to make a better informed decision about which locking approach to use.**

***Keywords*-component; formatting; style; styling;**

I. Introduction

In multi-threaded applications, synchronization overhead(SO) due to locks, is defined as accumulation of number of cycles threads wait while waiting for some other thread holding the lock to release it and subsequently waiting thread getting the lock. SO is higher for non-scalable locks(e.g. ticket lock) than queue based scalable locks(e.g. MCS, CLH) [1], due to the problem of exponentially increasing cache coherence overheads because of global spinning. This collapse in performance is even observed where moderate numbers if lock acquisition and release are involved.

Researcher [2] [3] have experimented would multiple locking algorithms on an array of benchmark application and kernels. Which locking algorithm would be best for which application is still an open problem. Simulation and analysis in this paper is confined to queue based locks and pthread mutex. Mutex is a hybrid lock which first attempts locking using simple atomic instructions before attempting high contention suitable MCS lock. Not all multi-threaded application suffer due to synchronization overhead.

A simple rule of thumb to gage extent of SO suffered by an application, is by taking product of number of lock allocation/de-allocation(N) made and size of critical section in cycles (C_{cs}), in every thread. Having large critical section i.e. locking at very course granularity is not advisable. So in most cases(well written applications), if N in a thread is large an application will suffer significant SO. SO is also affected by order in which waiting lock requests are served.

II. Queueing network model of synchronizing threads

Application with synchronizing threads are modeled as a closed queueing network. While Queueing network based modeling have been used to model multi-threaded application [4], synchronization overhead has been abstracted to an average delay. SO is a complex phenomena, increasing significantly in comparison to overall thread execution time as number of contending threads increases, too complex to be modeled as a constant delay. It is therefore felt, it should be modeled in detail.

Before modeling every multi-threaded application is instrumented and a trace of threads getting into and coming out of critical sections(CS) is generated. Based on these traces an application is classified as having homogeneous(each thread CS and non-CS(NCS) taking equal number of cycles) and heterogeneous(varying cycles for CS and NCS) threads. All details except threads experiencing contention due to locks are abstracted out. Each thread is thus reduce to a trace consisting of N time repetitive sequence of a set of critical sections and a set of non-critical sections, Fig.1.

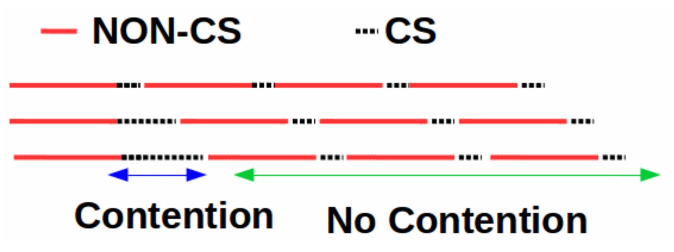

Figure 1. Threads as collection of repetitive NCS and CS sections

For the analysis, it is assumed that number of threads are less than or equal to number of cores. Also threads are pinned to their respective cores. Number of cores P defines the population size of the close queueing network. N, number of lock acquisition and release defines the number of iteration of the simulation. All queue's are modeled as /D/1 queues, with service time having exponential distribution. None of the existing tool have the desired feature to configure routing strategy from the infinite server NCS

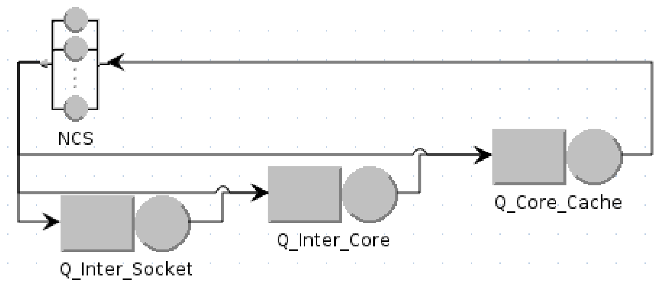

Figure 2. Queueing network having multiple queue's

delay node. This simulator defines the routing strategies that can be adopted by lock scheduler when multiple threads are concurrently trying to acquire the lock.

In simple case FCFS strategy can be followed. Another possibility is to allocate lock to threads with shortest remaining time (SRT). SRT can be defined by looking at whether requesting threads and thread holding the lock are on same core(Hyper-threads), requiring short intra-core delays, or are on the same socket, requiring longer inter-core(NOC) delays or on different sockets, requiring longest inter-socket delays. Starvation is a problem when following SRT policy, hence some aging policy is also implemented in the lock scheduler Fig.2.

III. Measurements and results

Simulation and analysis extracts parameters that would characterize SO for different applications.

A. Homogeneous threads

SO for homogeneous threads can be characterized using following parameters:

1) How many threads lock was assigned before it was reassigned to the current thread.
2) Aggregate of how many assignment of locks to other threads happened after last allocation to the a particular thread. This will show extent of load imbalance. For homogeneous threads it should be as low as possible.
3) Average latency of lock acquisition and release.

B. Heterogeneous threads

SO for Heterogeneous threads can be characterized using following parameters:

- Count of allocation to each delay node for different lock scheduling strategies.
- Average Latency of lock acquisition and release.

IV. Conclusion

It is observed that while SRT could be a good strategy for random heterogeneous threads, for homogeneous threads FCFS is an optimal strategy. In a queue its the variance that kills the performance not the latency.

References

[1] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proceedings of the Linux Symposium*, 2012, pp. 119–130.

[2] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pp. 33–48, 2013.

[3] H. Guiroux, R. Lachaize, and V. Quéma, "Multicore Locks : The Case is not Closed Yet," *USENIX Annual Technical Conference*, 2016.

[4] H. Che and M. Nguyen, "Amdahls law for multithreaded multicore processors," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 3056–3069, 2014.