Scalable Adaptive NUMA-aware Lock

Combining Local Locking and Remote Locking for Efficient Concurrency

Mingzhe Zhang
Francis C. M. Lau
Cho-Li Wang
Dept. Computer Science
The University of Hong Kong
{mzzhang,fcmlau,clwang}@cs.hku.hk

Luwei Cheng
Facebook
chengluwei@fb.com

Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

Abstract

Scalable locking is a key building block for scalable multi-threaded software. Its performance is especially critical in multi-socket, multi-core machines with non-uniform memory access (NUMA). Previous schemes such as local locking and remote locking only perform well under a certain level of contention, and often require non-trivial tuning for a particular configuration. Besides, for large NUMA systems, because of unmanaged lock server's nomination, current distance-first NUMA policies cannot perform satisfactorily.

In this work, we propose SANL, a locking scheme that can deliver high performance under various contention levels by adaptively switching between the local and the remote lock scheme. Furthermore, we introduce a new NUMA policy for the remote lock that jointly considers node distances and server utilization when choosing lock servers. A comparison with seven representative locking schemes shows that SANL outperforms the others in most contention situations. In one group test, SANL is 3.7 times faster than RCL lock and 17 times faster than POSIX mutex.

1. Introduction

Designing a scalable lock primitive for multi-core machines is a challenge, and continues to be so as the core count increases and memory hierarchies like NUMA get more complex. In large-scale NUMA machines, scalable locks need to avoid not only centralized contentions but also those that are across nodes.

There are in general two types of locks: local and remote. It has been shown that local locking (i.e., based on waiting on shared variables) is non-scalable and could have performance breakdown when the number of cores increases [1]. Although numerous designs have been proposed to try to improve local locking, such as many variants of spin lock [11, 12], MCS lock [9], RCU-based locks [2, 8], and hierarchical locks [3, 7], due to the intrinsic limitation of data sharing in the critical section, cache invalidation still happens frequently, especially when the lock contention is heavy.

Remote locking schemes use one dedicated server thread to handle all requests to access the critical section, and thus can significantly reduce cache bouncing. Examples include OyamaAlg [10], flat combining (FC) [5], H-Synch [4] and remote core locking (RCL) [6]. Using this method, however, clients need to communicate with the server via message exchange of which the overhead could sometimes outweigh the gain when the contention level is relatively low. Another issue in NUMA environments is how to avoid cross-node memory access as much as possible. Current approaches (e.g., [4]) only allow the server thread to accept the local node's lock requests, and the server thread will not be re-nominated on another NUMA node until all local requests have been processed. This leads to two issues: 1) if the local contention is very heavy, remote clients will starve; 2) if the local contention is very low on each node, the server may be re-nominated among all the NUMA nodes far too frequently, which damages cache locality.

To the best of our knowledge, there is currently no locking scheme capable of performing satisfactorily under *varying* contention levels. Therefore, an adaptive locking scheme that can properly and automatically adjust the synchronization method according to the contention level is necessary.

In summary, the main contributions of SANL are:

- A scalable synchronization scheme that switches adaptively between local locking and remote locking in multi-socket multi-core environments.

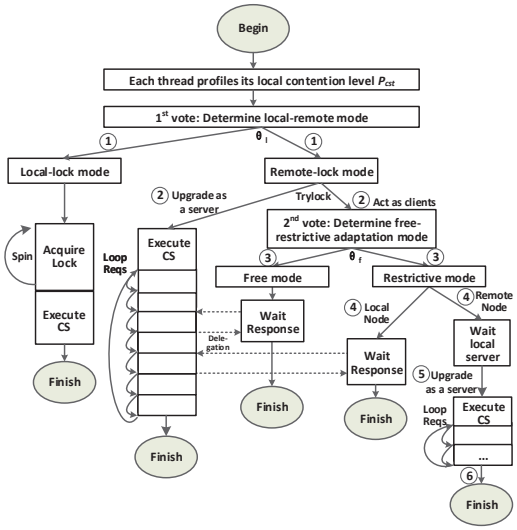

Figure 1: The execution flow of critical sections of SANL.

PPoPP '16, March 12-16, 2016, Barcelona, Spain

http://dx.doi.org/10.1145/2851141.2851176...$15.00

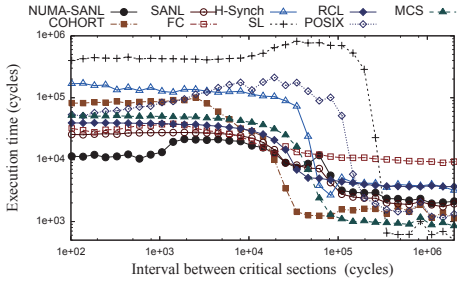

Figure 2: Execution times under different contention levels on the Intel machine (4 NUMA nodes). The number of shared cache line in the critical section is 1.

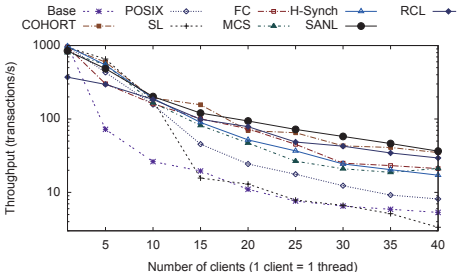

Figure 3: Throughput results of BerkelyDB's `StockLevel` transaction on the Intel machine.

- A NUMA-aware remote locking scheme that jointly considers node distances and server utilization.

2. Design

We design `SANL` to combine the benefits of local locking and remote locking. We also design a remote locking schemes with an efficient NUMA support to work with `SANL`. `SANL` improves existing dynamic remote locking scheme through 1) a request array together with an global *Id Manager* to avoid contention among the clients; 2) a new server downgrade policy to improve the server's utilization and reduce the frequency of server re-nomination. When dealing with NUMA in remote locking situations, `SANL` considers both node distances and the current server's utilization. In this way, both starvation of remote clients and frequent server re-nomination can be largely avoided. To achieve smooth adaptation, `SANL` adopts a voting scheme by profiling thread local contention level.

Figure 1 shows the general execution flow of `SANL`. When executing a critical section, a thread will profile its local contention level (denoted by C_l), and then vote for the global contention level (denoted by C_g). In Step ①, if the global contention level is below a threshold θ_l (which depends on the architecture), all threads will enter local locking mode; otherwise, they enter remote locking mode. In the remote-lock mode, one thread first executes *trylock* to attempt upgrading itself to the server thread (Step ②); the other unsuccessful threads will automatically become client threads. In NUMA environments, if the contention level is below a threshold θ_f, *free-mode* will be adopted which means all clients are allowed to send lock requests to the server thread, regardless whether they are in the same NUMA node or not (Step ③); otherwise, when the contention on the server node is sufficiently high, `SANL` will enter *restrictive-mode*: local clients are allowed to send requests while remote clients wait until the server node's contention level has come down or server re-nomination happens (Step ④). A thread can be in the server's role for only at most a limited iteration times T_s, and then it would downgrade to an ordinary thread to finish its own task. This ensures that the thread will not be occupied for too long, and server nomination can happen fairly among all the NUMA nodes. When server downgrading happens, if there are still unfinished lock requests, the corresponding clients will try to upgrade to a new server thread after a timeout of T_w in Step ⑤.

3. Preliminary Results

We have implemented `SANL` in Linux 3.14.3. We evaluate `SANL` with micro-level benchmarks and application benchmarks. The benchmark program includes a master thread that dynamically creates a set of of slaves to repeatedly contend for one critical section. The critical section includes acquiring lock, changing data in shared cache lines and releasing lock. After all contending threads have terminated, the master thread will compute the throughput results. The *contention level* is controlled by varying *the interval between accessing the critical section* per thread: the shorter the interval, the higher the contention. In micro-benchmark, we vary the interval from 100 to 2,000,000 cycles. We apply `SANL` to three popular multi-threaded applications: BerkeleyDB with TPC-C, Memcached and Phoenix2 suite.

Performance results on a 40-core Intel machine show that `SANL` can adapt to various contention levels satisfactorily. In one micro-benchmark, `SANL` is 3.7 times faster than RCL and 17 times faster than POSIX lock under high contention, shown in Figure 2. In the Berkeley DB, Figure 3 shows the result of StockLevel transaction. `SANL` achieves performance improvements of up to 66% over RCL and 9.5 times over POSIX lock. In Memcached tests, `SANL` is 24% faster than RCL and 3.8 times faster than POSIX lock. For Phoenix2, `SANL` also consistently ranks the highest among other locks, with a speedup of up to 58% over POSIX lock.

References

[1] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proc. Linux Symposium*, 2012.

[2] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proc. ASPLOS*, 2012.

[3] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proc. PPoPP*, 2012.

[4] K. Fatourou, Panagiota and Nikolaos. Revisiting the combining synchronization technique. In *Proc. PPoPP*, 2012.

[5] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. SPAA*, 2010.

[6] J.-P. Lozi, F. David, G. Thomas, J. Lawall, G. Muller, et al. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proc. USENIX ATC*, 2012.

[7] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *Proc. ICPP*, 2006.

[8] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Proc. AUUG*, 2001.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[10] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proc. PDSIA*, 1999.

[11] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proc. HPCA*, 2003.

[12] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proc. PACT*, 2010.