Scalable Adaptive NUMA-Aware Lock

Mingzhe Zhang, Haibo Chen, *Senior Member, IEEE*, Luwei Cheng, Francis C. M. Lau, *Senior Member, IEEE*, and Cho-Li Wang, *Senior Member, IEEE*

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 in-place locks and delegation locks only perform well under a certain level of contention, and often require non-trivial tuning for a particular configuration. Besides, in large NUMA systems, current delegation locks cannot perform satisfactorily due to lack of optimized NUMA policies. In this work, we propose SANL, a locking scheme that can deliver high performance under various contention levels by adaptively switching between in-place locks and delegation locks. To optimize the performance of delegation locks, we introduce a new NUMA policy that jointly considers node distances and server utilization when choosing lock servers. We have implemented SANL and evaluated it with four popular multi-threaded applications (Memcached, Berkeley DB, Phoenix2 and SPLASH-2), on a 40-core Intel machine and a 64-core AMD machine. The comparison results with seven other representative locking schemes show that SANL outperforms them in most contention situations. For example, in one group test, SANL is 3.7 times faster than RCL lock and 17 times faster than POSIX mutex.

Index Terms—Delegation lock, adaptive synchronization

1 Introduction

Designing a scalable lock primitive for multi-core machines is challenging, especially when there is a large number of cores and when the memory hierarchy is complex which is typical of many NUMA machines. In large-scale NUMA machines, to achieve scalable performance, locking should try to avoid centralized contention and meanwhile preserve memory-access locality across NUMA nodes as much as possible.

In general, there are two common types of locks: *in-place locks* and *delegation locks*. In this paper, we define *in-place locks* as those that are implemented by waiting on shared variables before entering the critical section (CS), such as spin lock (SL). It has been shown that waiting on shared variables is non-scalable and could experience performance breakdown when the number of cores increases [1]. Although numerous designs have been proposed to try to improve in-place locks, such as the many variants of spin locks [2], [3], [4], [5], MCS lock [6], RCU-based locks [7], [8], and hierarchical locks [9], [10], due to the intrinsic feature of data sharing in the critical section, cache invalidation still occurs rather frequently, especially when the lock contention is heavy. David et al. [11] also point out that the performance of in-place locks can be largely affected by the hardware architecture as well as the runtime contention level.

- *M. Zhang, F.C.M. Lau, and C.-L. Wang are with the Department of Computer Science, University of Hong Kong, Hong Kong. E-mail: {mzzhang, fcmlau, clwang}@cs.hku.hk.*
- *H. Chen is with the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Minhang, Qu 200240, China. E-mail: haibochen@sjtu.edu.cn.*
- *L. Cheng is with Facebook, Menlo Park, CA 94025. E-mail: chengluwei@fb.com.*

Manuscript received 13 Dec. 2015; revised 19 Oct. 2016; accepted 19 Oct. 2016. Date of publication 18 Nov. 2016; date of current version 17 May 2017. Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2630695

Delegation locks periodically or stably rely on one dedicated server thread to handle all requests to access the critical section, and thus can avoid the problem of cache bouncing to a large extent. Examples include OyamaAlg [12], flat combining (FC) [13], P-Sim [14], H-Synch [15] and (RCL) [16], [17]. However, delegation locks are not without issues: since clients need to communicate with the server via message exchange, the overhead can sometimes outweigh the gain when the contention level is very low. Lozi et al. [16], [17] show that on a 48-core machine with four 12-core Opteron 6,172 processors, when a program spends less than 70 percent of its execution time inside critical sections, in-place locks such as MCS lock and POSIX mutex are more efficient; however, when the time spent in critical sections is larger than 70 percent, delegation locks such as RCL start to scale better. Although the threshold point may vary with different hardware platforms, it is evident that each type of locking scheme has its own comfort zone in which it would scale well, and one cannot completely replace the others. This echoes the conclusion of a recent study on multi-core synchronization that "*every locking scheme has its fifteen minutes of fame*" [18].

Another issue that delegation locks may face in NUMA environments is the amount of cross-node memory accesses which, for the sake of performance, should be avoided as much as possible. One possibility is to make clients only send their requests to their local node, similar to the contention control technique in H-Synch [15]. But since there is only one server thread (combiner), clients in a remote core will have to wait until the server thread leaves its current core (downgrades to a normal thread) and then re-nominated on the remote core, which could mean a long synchronization delay. When the server's utilization is low, server re-nomination may happen too frequently among different NUMA nodes, which can lead to serious cache invalidation.

To address the above challenges, this paper proposes SANL, a locking approach that: 1) combines the in-place

1045-9219 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

mode and the delegation mode and their respective benefits, and 2) optimizes the NUMA policy in delegation mode to better utilize the server and reduce remote clients' waiting times. SANL is highly scalable as it can adaptively and dynamically switch between the two locking schemes according to the measured contention level.

To appropriately adapt between the in-place mode and the delegation mode, SANL adopts a simple voting scheme. When dealing with NUMA in delegation mode, SANL considers both node distances and the current server's utilization: rather than letting each client wait passively until the server is nominated on their local nodes, SANL allows some clients to send requests to the remote server when the server is not facing high contention. In this way, both starvation of remote clients and frequent server re-nomination can be largely avoided. To make SANL portable over different hardware platforms, we add an easy-to-use profiler to automatically determine the proper threshold for the adaptation.

We implement SANL in software for the commodity x86 multicore architectures and tested it with a set of popular multi-threaded applications, including Memcached [19], Berkeley DB [20], Phoenix2 [21] and SPLASH-2 [22], [23]. We have released SANL's source code, and the testing tools and results at https://github.com/SANL-2015.

We evaluate SANL with both micro-level benchmarks and application-level benchmarks. Performance results on a 40-core Intel machine and a 64-core AMD machine show that SANL can satisfactorily adapt to various contention levels. In one of our micro-benchmarks, SANL is 3.7 times faster than RCL and 17 times faster than POSIX lock under high contention. Under low contention, SANL performs close to in-place locks while shows some management overheads. In the Berkeley DB and Memcached tests, SANL achieves performance improvements of up to 66 and 33 percent over RCL, and 9.5 and 3.8 times over POSIX lock respectively. In Phoenix2 and SPLASH-2 tests, SANL also consistently outperforms the other locks, with a speedup of up to 58 percent over POSIX lock.

The main contributions of this paper are:

- A scalable synchronization scheme that switches adaptively between the in-place mode and the delegation mode in multi-socket multi-core environments.
- A NUMA-aware delegation lock scheme that jointly considers node distances and server utilization.
- A set of evaluations that confirm the effectiveness of SANL using micro- and macro-benchmarks.

2 Background and Motivation

2.1 In-Place Lock Synchronization

Naive spin lock is the simplest in-place lock: all threads use an atomic exchange primitive to keep spinning on a *global* lock variable until it becomes available. Naive spin locks cannot guarantee fairness. To solve the problem, ticket lock records the sequence of locking requests, and has been implemented in the Linux kernel. Naive spin locks also suffer from a significant amount of cache invalidation traffic. Agarwal et al. [24] propose a class of adaptive backoff methods that can significantly reduce the memory traffic due to accessing synchronization variables. Radovic et al. [2] propose HBO to further improve the backoff scheme for NUMA environments by favoring local threads, which can substantially reduce cross-node memory accesses. Vasudevan et al. [3] propose biased-lock with lock priority to improve cache efficiency.

A common problem of all these spin locks is that all threads contend on a *single* memory location, which could generate excessive cache invalidation and bouncing. To mitigate this issue, Mellor-Crummey et al. [6] propose MCS lock: each thread lets its lock request join a *request queue* and spins only on its own variable until the previous thread hands over the lock. Auslander et al. [25] introduce the K42 lock to improve MCS lock's compatibility with legacy code. K42 lock is achieved by leveraging on-stack information, which requires fewer API changes.

POSIX mutex lock (*pthread_mutex_lock*) adopts a different approach: rather than busy-waiting when the lock is unavailable, it makes the lock-acquiring thread sleep until being woken up by the lock-holding thread. However, since sleeping induces additional context switches, it is suggested that POSIX mutex should only be used with a lengthy critical section.

There has also been much work in improving reader/writer locks (rwlocks) that are designed for read-mostly scenarios [26], [27], [28], [29]. RCU-based locks [7], [8] eliminate contention among readers through a fence-free reader-side critical section and by copying the shared data during updating. Arbel et al. [30] described several PRCU implementations to achieve good trade-offs between read overhead and short wait-for-readers time. Calciu et al. [31] present a novel family of reader/writer locks that are designed to leverage NUMA features. However, RCU places several constraints in the usage such as single-pointer update and read-once semantics.

Hierarchical locks are tailored to fit today's NUMA architectures. Luchangco et al. [9] introduce HCLH, a hierarchical version of the CLH queue-lock [32] with an in-place lock queue for each NUMA node. Dice et al. [10] design a general technique called "lock cohorting" to transform in-place lock algorithms into NUMA-aware ones. Chabbi et al. [33] further improve lock cohorting as HMCS for multi-level NUMA systems. Based on HMCS, Chabbi et al. [34] design an AHMCS lock with adaptive level HMCS and hardware transactional memory, which seem to perform well under a broad range of contention levels. Hierarchical locks show good throughput, since synchronization local to a NUMA node is faster than global synchronization. Critical sections executed on the same NUMA node can reuse shared variables that are stored in their common caches. While hierarchical locks offer good performance, they are generally based on traditional lock algorithms. Delegation locks can maintain the shared variables in the server core with theoretically higher data locality than hierarchical locks. According to on our experiments, SANL shows better improvement for NUMA with higher utilization of servers than directly applying the lock cohorting technique on delegation locks like H-Synch [15].

Usui et al. [35] propose an adaptive locking technique (AL) to dynamically switch between transaction and mutex locks according to the observed behavior of the critical section, e.g., number of blocked threads and transaction retries. Dice et al. [36] introduce the ALE library to integrate transactional locks using hardware transactional

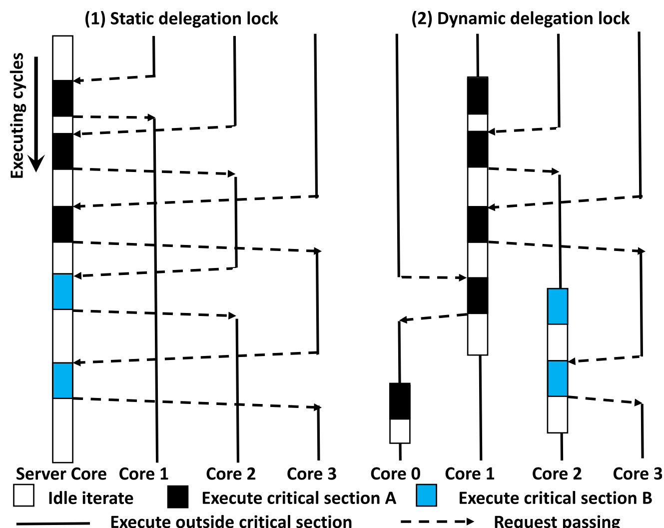

Fig. 1. Execution flows of delegation lock: Static versus dynamic.

memory and optimistic execution of operations in software. This library provides three adaptation modes (HTM, software optimization, and lock modes) and two policies (static and adaptive) to help determine the adaptation values. Evidently, the research community is increasingly interested in this idea to achieve adaptive concurrency control in applications. We also think the idea is promising, and believe that it can help achieve higher performance through adaptation with delegation lock in NUMA environments.

2.2 Delegation Lock Synchronization

Delegation lock schemes periodically or stably rely on one dedicated server thread which executes the critical section on behalf of all other threads. Since only the server thread accesses the critical section, data locality is better preserved in the cache. Delegation locks can be further categorized as static or dynamic, and their differences in execution are illustrated in Fig. 1. In static delegation locks, one server thread serves all lock instances, whereas, in dynamic locks, each lock instance dynamically upgrades a normal thread to serve as its server.

RCL [16], [17] is a static delegation lock designed for high-contention situations. However, the static method can cause false serialization when one server core is assigned to multiple lock instances. For example, in Fig. 1 1, core 0 acts as the server for both critical sections A and B; though there is no contention between the two lock instances, all requests must be sequentially executed. Another problem is that excessive computing resources will be occupied if too many servers have been assigned, which can also lead to performance degradation.

CPHASH [37] is a scalable, concurrent hash table for key/value caches implemented for the static delegation lock technique. Basically, it partitions the hash table across static server cores, and clients perform operations on a partition by sending a message through shared memory to the right partition. CPHASH achieves 1.6 to 2 times higher throughput than LOCKHASH, an optimized fine-grained locking implementation. However, a limitation of CPHASH is that the optimization is specific to hash tables.

In dynamic delegation locks (e.g., [12], [13], [14], [41]), the server thread is launched on a per-lock rather than per-core basis; once the server thread has processed all the requests of its lock instance, it will downgrade to a normal thread. The execution flow is illustrated in Fig. 1 2. OyamaAlg [12] maintains the lock requests in a linked-list LIFO queue, with queuing contention at the head node. Flat combining [13] reduces this contention by enforcing a FIFO processing order. P-Sim [14] further improves it with an array-based design and an atomic *Fetch-and-Add* operation that manipulates the array index. Dice et al. [38] apply the idea of FC to MCS lock to optimize the lock request queuing operation; however, similar to in-place lock, a thread holding the lock would execute the critical section on its own. To make delegation locking easy to use, David et al. [41] provides libraries for C and C++.

Fatourou et al. [15] revisit the combining technique and propose CC-Synch, as well as a hierarchical version called H-Synch to take into account the CPU topology of NUMA systems. Using H-Synch, the threads belonging to the same NUMA node are combined as a cluster like lock cohorting [10], which is protected by an extra queue lock. However, hierarchical delegation locks can suffer performance degradation due to low server utilization.

Petrović et al. [39] study how to further improve the performance of delegation over cache-coherent shared memory by considering the subtleties of the underlying cache coherence protocol, and then propose two optimizations techniques: backoff in local-spin and weakly-ordered streaming stores.

2.3 Motivation

While there has been much work aiming at improving the performance of in-place locks, they share the same limitation: all threads must fetch the shared data to their own CPU caches when executing the critical section, making cache bouncing a real issue.

Delegation locks on the other hand maintain data locality in the cache of the server core, but they require message exchange between the server thread and client threads, which could translate into significant overhead. Lozi et al. [16], [17] has shown that when lock contention is low, the management overhead of delegation locks could be even more pronounced than the cache bouncing overhead of in-place locks. Calciu et al. [40] have the same conclusion in their survey of three data structures for delegation lock implementation: *MPSCChannel*, *InletQueue*, and *DNCInletQueue*. They conclude that delegation locks can sometimes outperform in-place locks, particularly when enough data is accessed which ensures that the benefits of delegation lock outweigh its communication costs; nonetheless, when critical sections are short, some in-place locks perform substantially better than delegation locks. To briefly compare in-place locks with delegation locks, we summarize the features of existing representative locks in Table 1.

The use of delegation locks in NUMA environments presents a challenge regarding performance optimization. In delegation lock schemes, without NUMA policies [13], [16], [17], both remote clients and local clients would send requests to the server thread. Obviously, sending remote requests take longer time than sending local requests. To be NUMA-aware, Fatourou et al. [15] only allow the server thread to accept the local node's lock requests, and the

TABLE 1
Comparison of Representative Lock Primitives and SANL

	SL	Backoff	HBO	Ticket	POSIX	MCS	K42	PRCU	HCLH	Cohort lock	AL	ALE	FC	FC-MCS	H-Synch	RCL	SANL
Low racing on lock variables													✓	✓	✓	✓	✓†
Good shared data locality													✓		✓	✓	✓†
Conditional wait support	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓
NUMA-aware			✓						✓	✓				✓	✓		✓
Apply adaptive scheme		✓	✓								✓	✓					✓
Unchanged lock fairness				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
low code complexity	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓						
Number of tuning parameters	0	1	2	0	0	0	0	1	1	1	3	2	1	1	2	0	3

† *Delegation-lock mode of SANL under high contention for critical sections.*

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 get re-nominated too frequently among all the NUMA nodes such that it hurts cache locality.

To the best of our knowledge, there exist currently very few locking schemes that are capable of performing satisfactorily under *varying* contention levels. Therefore, an adaptive locking scheme that can appropriately and automatically adjust the synchronization method according to the contention level is desired.

3 The Design of SANL

We craft SANL to combine the benefits of in-place locks and delegation locks. We also extend existing delegation locks with an efficient NUMA support to work with SANL.

Fig. 2 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 in-place mode; otherwise, they enter delegation mode. In the delegation mode, one thread first executes *trylock* to attempt to upgrade itself as 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 of whether they are in the same NUMA node (Step ③) or not; otherwise, when the contention on the server node is sufficiently high, SANL will enter *restrictive-mode* (Step ④), where 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 occurs (Step ⑤). A thread can be in the server's role for only at most a limited number of times T_s, and then it will downgrade to a normal thread to finish its own task. This ensures that the thread will not be occupied for too long, and server nomination can take place fairly among all the NUMA nodes. When server downgrading happens, if there are still unfinished lock requests, the corresponding clients will try to upgrade themselves as a new server thread after a timeout of T_w in Step ⑦. If a remote client detects the remote server node's contention level has come down or a local server node appears, the client will delegate its requests to the server (Step ⑥). For convenience, the definitions of the symbols we use can be found in Table 2.

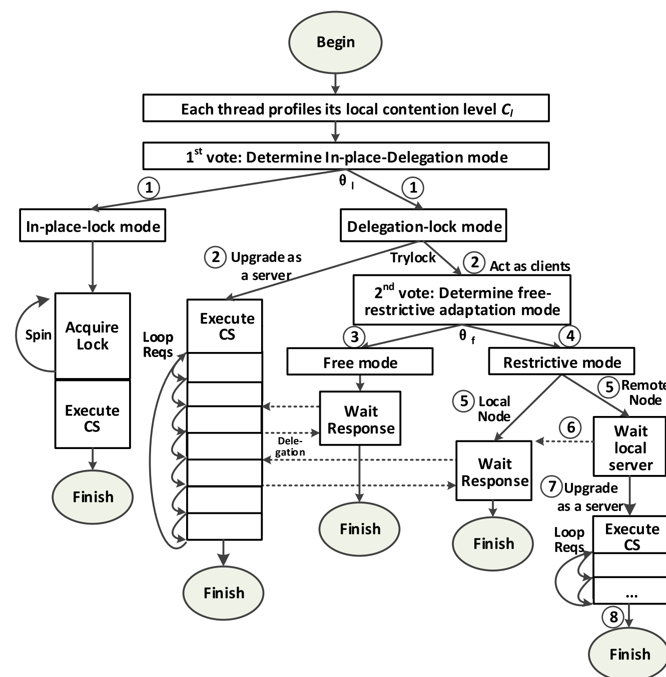

Fig. 2. The execution flow of critical sections of SANL.

3.1 Adaptation Scheme

Under low contention, in-place locks are more efficient, so SANL switches to the in-place mode; when lock contention is high, SANL switches to the delegation mode. In order to determine when to switch, SANL relies on the global contention level C_g, which can be derived from each thread's local

TABLE 2
The Descriptions of the Symbols We Use in SANL

Symbols	Descriptions
θ_l	The threshold of *In-place-Delegation* mode transition
θ_f	The threshold of *free-restrictive* mode transition
L_s	Dynamic limit of the server's iteration times
T_s	Static limit of the server's iteration times
T_w	Maximum spinning times of client threads
C_l	The percentage of time executing critical sections for a given thread
C_g	The global contention, voted from all contending threads
Cnt_i, Cnt_d	Cumulative counters about whether C_l is smaller than θ_l
Cnt_r	Cumulative counter about requests served in one serving iteration
Cnt_d	Cumulative counter about whether *RequestDensity* is low

contention level C_l. C_l is defined as the ratio of time in critical sections over a certain past period:

$$C_l(i) = \alpha \, * \, \frac{CS(i-1).end - CS(i-1).begin}{CS(i).begin - CS(i-1).begin} + \, (1 \, - \, \alpha) \, * \, C_l(i-1), \tag{1}$$

where α is a weight factor (0.5 by default), and $CS(i)$ and $CS(i-1)$ represent the current and the previous critical section respectively. $(CS(i-1).end - CS(i-1).begin)$ is the time spent in last CS. $(CS(i).begin - CS(i-1).begin)$ is the time spent between last CS and current CS. The result of the division represents the ratio of time spent in CS. In order to appropriately adjust the contention level, C_l considers the new ratio and former ratio together with a weight factor α. The equation applies floating point computation. In both Intel and AMD machines, the multiplication takes three cycles and division takes about 27 cycles. For short critical section under high contention, the floating point computation only accounts for 0.13 percent execution time of a critical section. While under low contention, it is non-trivial (6.25 percent of a critical section on average). For applications with short critical sections under stable low contention, we can configure SANL to compute $C_l(i)$ every N (e.g., 3) critical sections to reduce this overhead. For other cases, we suggest to make SANL compute $C_l(i)$ every critical section because the overhead of the floating point computation is acceptable whereas sampling computation could lead to inaccurate computation of contention and degrade the performance.

Algorithm 1. SANL Voting Algorithm

Global variables: int $C_g = 0$;
Per-thread variables: int $Cnt_i = 0$, $Cnt_d = 0$; bool local_vote = false;

1 **if** local_vote **then**
2   **if** $C_l < \theta_l$ **then**
3   increment Cnt_i; /* Buffer to reduce thrashing */
4   **if** $Cnt_i >$ Boundary **then**
5   local_vote = FALSE; /* Support in-place mode */
6   atomic decrement C_g;
7       $Cnt_i = 0$;
8   **else if** $Cnt_i > 0$ **then**
9   decrement Cnt_i;
10 **else**
11   **if** $C_l \geq \theta_l$ **then**
12   increment Cnt_d; /* Buffer to reduce thrashing */
13   **if** $Cnt_d >$ Boundary **then**
14   local_vote = TRUE; /* Support delegation-lock mode */
15   atomic increment C_g;
16       $Cnt_d = 0$;
17   **else if** $Cnt_d > 0$ **then**
18   decrement Cnt_d;

Ideally, the value of C_g should be the average of all C_l values, but such a global computation could be expensive. Another problem is that if the two locking schemes perform similarly when the contention lingers around a certain level, threads may thrash between two modes. To reduce the computing overhead of C_g and to tolerate the sudden deviation in contention level, SANL adopts a voting scheme, treating C_g as a global vote value from all contending threads. As described in Algorithm 1, for a given thread, if its local contention level C_l is below a pre-determined threshold θ_l (Line 2), it will increment its counter (Line 3); if the counter value exceeds a boundary, the thread begins to vote for the in-place mode (Lines 4-7). It uses *fetch-and-add* atomic instruction to update C_g. The steps of voting for the delegation mode are similar (Lines 11-18). This design effectively accommodates the sudden changes of C_l and reduces the atomic operations on C_g. When the global vote C_g is greater than half of the number of contending threads, threads will enter the delegation mode; otherwise, they stay in the in-place mode. Every thread decides the lock mode depending on the temporal C_g, requiring no global barrier for transition. Fig. 3 shows an example of execution timeline for Algorithm 1. Suppose there are three threads and five stages (for simplicity, in each stage, the sequence of execution is $T1 > T2 > T3$), and suppose the boundary is 1: each thread votes for C_g depending on C_l; during stage 2 and 5, due to C_g's change, SANL adapts between in-place mode and delegation mode. As SANL adopts Algorithm 1, C_g can tolerate sudden deviation in contention level. The scenarios of coexisting lock modes like stage 2 only happen in lock mode adaptation when the contention stably changes. Although different threads can choose different lock modes,

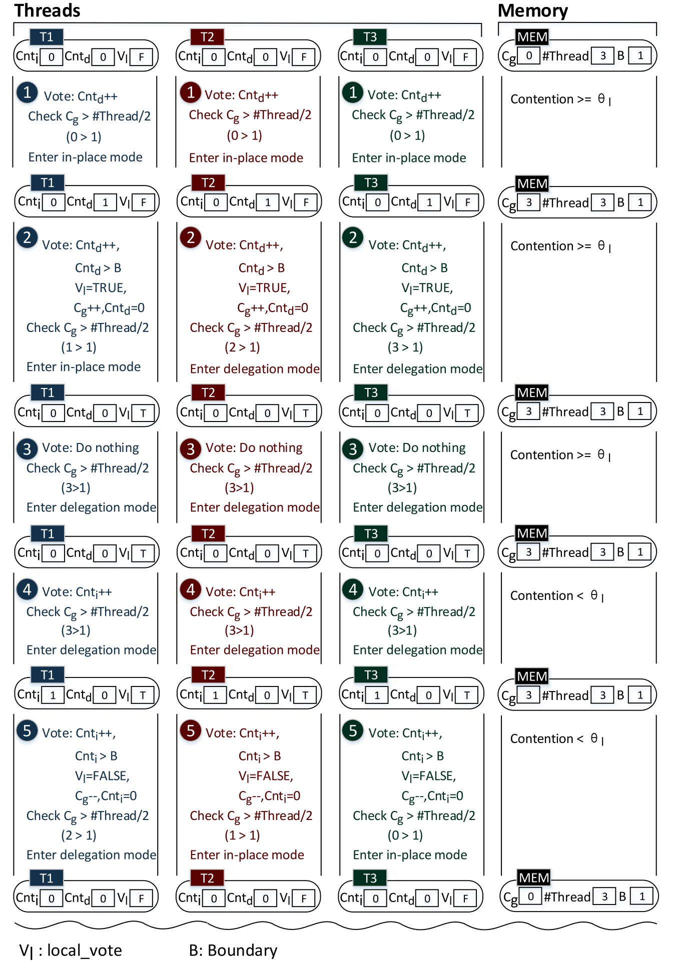

Fig. 3. An example of SANL voting algorithm.

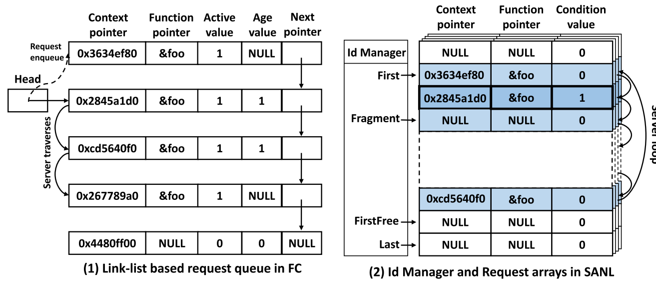

Fig. 4. Comparison between FCQueue in FC and request array in SANL. Every instance of SANL has a request array and they share a global Id manager.

SANL is always safe. Mutual exclusion is guaranteed even for such cases as when there are both in-place-mode and delegation-mode threads (stage 2 and 5): when an in-place-mode thread holds the lock, all delegation-mode threads repeatedly *tryLock* until a server thread appears (after the lock-holder leaves the critical section). When the lock is held by a delegation-mode thread (server), all in-place-mode threads will wait until the server thread releases it.

3.2 SANL's Delegation Lock

Previous dynamic delegation locks such as FC [13] and H-Synch [10] adopt linked-list based queueing to manage client requests. Fig. 4 1 shows the design of FC's request queue (FCQueue), which is a FIFO-like scheme to avoid the synchronization overhead between the server and the clients. Each thread can upgrade itself as a server via a successful CAS operation, while the other unsuccessful threads enqueue their requests to the head and update themselves as the head node. The server traverses the FCQueue from head to tail, conducts all non-null requests, sets the age of each of these requests to the current count, and sends responses to the corresponding clients. If the age indicates that a cleanup is needed, the server will traverse the FCQueue to remove all the requests whose ages are older than a certain value. Unfortunately, FC needs to actively insert request nodes into the queue. Although FC avoids the synchronization overhead between the server and the clients, the clients still have to contend with each other to enqueue their requests into FCQueue, which could be a performance bottleneck. In addition, the server traverses FCQueue from head to tail for a fixed number of retry times. Without dynamic policies, this could waste serving time under low contention. Under high contention, this limits the server utilization. Thus, server downgrading and re-nomination can happen very frequently in FC, generating much management overhead. To overcome the above limitations, SANL improves dynamic delegation lock with two approaches. First, a request array together with a global *Id Manager* are used to avoid contention among the clients. Second, we introduce a new server downgrade policy to improve the server's utilization and reduce the frequency of server re-nomination.

Fig. 4 2 illustrates the design of the request array. The request array is a per-lock data structure. The array size is a constant value corresponding to the maximum allowed number of clients (256 × the number of cores, the same as RCL's). The *Id Manager* is a global data structure for all threads which records and supervises all request arrays' information. It maintains a bitmap with the same size as a request array to record active indices of threads. When a contending thread comes around, the *Id Manager* assigns the thread a free index from the bitmap and updates the bitmap. When a thread exits, the corresponding index will be recycled by the *Id Manager*. Hence, the *Id Manager* can easily compute the number of contending threads from the bitmap. The *Id Manager* scans the bitmap only when new threads are created and the size of the bitmap is a sufficiently large constant. In this way, no thread needs to busy-scan the bitmap for an available index.

The indices `First` and `FirstFree` represent the beginning of the active and inactive request arrays respectively, while `Last` always points to the end of the request array. Since each client has a unique index in the request array, it can directly communicate with the server without additional lock operations. In this way, the contention for the request array among clients can be gracefully avoided. Every thread can upgrade to a server through a successful *trylock* and then iterate over the requests to check the critical section pointers and execute them on behalf of the other threads. To support conditional functions, the lock request includes a conditional wait variable. The server updates this variable to notify the client that its request is being handled. Thus the client waits for the response without timeout. We are aware that RCL [16], [17] and P-sim [14] also have a similar request array design. However, since RCL enforces a static server thread, it poses the problem of false serialization (as has been discussed in Section 2.2). In P-sim, since the server does not have pre-assigned indices for the threads, each thread has to acquire an index through a *Fetch-and-Add* atomic primitive, which may cause contention among the clients with extra overheads.

In order to avoid the server frequently switching between different threads, SANL introduces *RequestDensity* for the server to manage its downgrade time, which is defined as

$$RequestDensity = \frac{num_of_requests}{num_of_contending_threads}, \qquad (2)$$

which can be easily calculated when a server iterates over the request array. If *RequestDensity* is less than or equal to a threshold (zero by default), the server switches to a normal thread. To deal with sudden fluctuations of request density, server downgrade will not happen immediately when low request density is detected. We define L_s to represent the threshold on the number of times that request density is detected low. Each time the server iterates over the request array, it would recompute *RequestDensity* and then increment or decrement the corresponding counter's value. The server downgrades to normal thread only when the counter exceeds L_s. In fact, L_s is related to the contention level of each thread C_l. It also influences the serving time. If L_s is too large, a server may waste time waiting for just a few requests, whereas if L_s is too small, the server switches too frequently. SANL provides a profiler to automatically determine the proper value for L_s, which is introduced in Section 3.4.3.

3.3 SANL's NUMA Policy

Prior work [15] tends to limit contention in the local node. According to this, we implement a similar *distance-first* policy

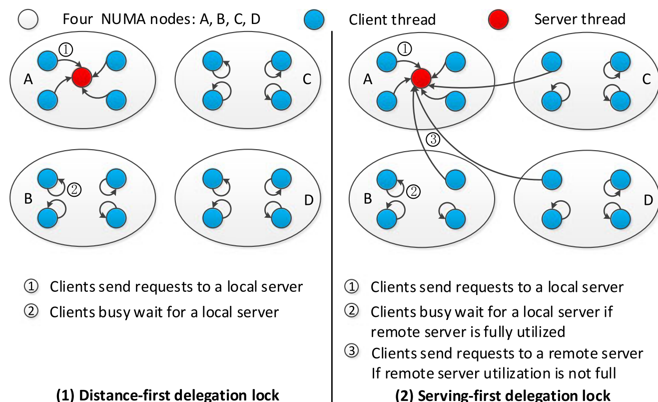

Fig. 5. Distance-first delegation lock and serving-first delegation lock.

in SANL: client threads are only allowed to send requests to their local servers in order to avoid the penalty of cross-node memory access, as illustrated in Fig. 5 1's node A. If a client is far away from a server, it keeps waiting under the dynamic server nomination scheme, such as clients in nodes B, C and D. Only when the server in node A has downgraded will another thread in another node upgrade to serve its local requests. In the distance-first scheme, the server thread adopts the downgrade policy presented in Section 3.2 with *RequestDensity* computation and proper L_s configuration. When node-local contention is low, the distance-first scheme suffers from one of two limitations: 1) the server thread spends a lot of time idle waiting for node-local requests while a remote client starves (the same case as L_s is too large); 2) if a server downgrades itself because it receives too few requests, frequent downgrading and nomination can cause frequent cache invalidation of shared data in critical sections (the same case as when L_s is too small).

3.3.1 Serving-First Policy

Based on SANL's dynamic delegation lock, we design a *serving-first* policy to deal with the problems of the *distance-first* policy. The idea is illustrated in Fig. 5 2. Basically, a local client would always send requests to its local server if one exists, like clients in node A. For a remote client, a second voting is applied to determine whether it should send requests to the current server, like some of the clients in nodes B, C and D. The voting method is the same as that in Algorithm 1. In the voting process, we define another threshold θ_f to compare with each client's thread-local contention level C_l. If half of the clients' C_l values are below θ_f, which means the global contention is not very high, the server will accept requests from all remote clients; we call this *free-mode*. Otherwise, in the second case, only the remote clients with lower C_l than θ_f will be admitted, which we call *restrictive-mode*. By improving the server's utilization, SANL avoids the situation that the server would frequently downgrade when the global contention is mild, and meanwhile, reduces the remote clients' starvation.

3.3.2 Execution Flow of Server and Client

SANL nominates a server through two methods in a dynamic manner. First, a client thread will automatically become the server via a successful *trylock*. Second, SANL maintains a FIFO queue Q, a standard lock-based linked-list, to monitor the remote clients; if Q is not empty, the server will hand over its role to the first thread in Q when it downgrades; otherwise, a server releases SANL directly. A client will enqueue itself to Q only when two conditions are satisfied: 1) a client has been waiting for a local server for a certain time T_w; 2) there are no local threads in Q. If a local thread is already in Q, it is guaranteed that after a certain time,[1] SANL will nominate a local server. To ensure only one thread of a NUMA-node can join the queue, each node has a node-CAS-lock. A thread must successfully acquire the node-CAS-lock before joining. The node-CAS-lock effectively limits the contention to only the local NUMA-node, without impacting the server's performance.

The server-side algorithm is shown in Algorithm 2, adopting *PAPI* library. A server iterates through the request array to finish delegations for at most T_s times and profiles the request density to determine whether it should finish serving (Fig. 2⑧). A special case of a server would be if it blocks on conditional variables in a critical section. Every time a server handles a request, it first updates *cond* (Line 8) to notify the corresponding client. When a critical section calls a SANL conditional wait function, the server downgrades itself, acquires a POSIX mutex lock and executes the POSIX conditional wait function. After being woken up by a signal, the thread releases the POSIX mutex lock, upgrades itself as a server and continues serving the requests. During conditional wait, another thread can upgrade to become a new server.

The client-side algorithm is shown in Algorithm 3. A client first conducts the second voting to make the NUMA decision of serving-first delegation lock. It checks both local and global votes. If a client has waited for more than T_w times, it follows the dynamic server nomination scheme to upgrade to a server Fig. 2⑦ During the upgrade (Lines 16-18), if a server has handled the client's request and downgraded to a normal thread, the client does not need to execute its own critical section again.

3.4 Other Considerations

3.4.1 Ensuring Responsiveness

Since every server in the delegation mode is upgraded from a normal thread, it is important that the thread does not act as the server forever. Besides, a client needs the guarantee that it never waits too long in order to enter the critical section. There are two strategies in SANL to ensure responsiveness.

A Server Never Serves for Too Long. An application may have its custom maximum response time. Therefore, a server should downgrade and revert back to a normal thread within the required response time. For example, in Memcached, if a thread does not respond after a certain time, Memcached will throw a timeout exception. SANL provides T_s as a tunable threshold to ensure a server would download appropriately. In our evaluation, by setting T_s with our default value, even under the highest contention, we find that the average serving time of the server thread is less than one millisecond, which should have a negligible impact on most applications' response times.

1. At most $T_s \times$ max time of iterating the request array once $\times$ index in Q.

Algorithm 2. The Execution of SANL's Server Thread

```
   structures:
      lock_t: {int server_state, request_t* request_array,
      locallock* ll};
      request_t: {void* code, void* context, int cond};
      profiler_t: {long start, long end};
   Input variables: lock_t* gl; void* code, * context;
   Global variables: queue* Q;
   Per-thread variables: request_t* last, * req; profiler_t* p;
 1 p.start = PAPI_get_real_cyc();
 2 res = code(context)            /* Execute own critical section*/
 3 gl→server_state = UP           /* Update server state */
 4 while gl→server_state = UP do  /* Iteratively serve
                                     requests of clients */
 5   last = gl→request_array[firstFree];
 6   for req = gl→request_array[first] to last do
 7     if req→code then
 8       req→cond = TRUE;
 9       req→context = req→code(req→context);
10       req→code = NULL;
11       req→cond = FALSE;
12       Cnt_r++                  /* Update request counter */
13   Update request density and density counter Cnt_d;
14   T_serving++;
15   Cnt_r = 0;
16   if Cnt_d > L_s || T_serving > T_s then  /* Check density
                                     counter and serving times */
17     gl→server_state = DOWN;
18 if Q not empty then
19   hand gl→ll to the head thread of Q;
   else
20   unlock(gl→ll);
21 p.end = PAPI_get_real_cyc();
22 return res;
```

Algorithm 3. The Execution of SANL's Client Thread

```
   Input variables:         lock_t* gl; void* code, * context;
   Global variables:        int C_g; queue* Q; int N_t; /* Number
                            of threads */
   Per-thread variables:    request_t* req; profiler_t* p; bool
                            local_vote;
 1 p.start = PAPI_get_real_cyc();
 2 vote for free-restrictive adaptation; /* Make a decision
                       between free and restrictive modes */
 3 while local_vote && C_g > N_t/2 && isRemoteServer() do
 4   T_wait++;
 5   if T_wait > T_w then
 6     if No local threads in Q && enqueue successfully then
 7       lock(gl→ll)          /* Wait a server to hand lock */
 8       dequeue(Q);
 9       upgrade as server;
10 req = gl→request_array[selfId]  /* Send request to server */
11 req→context = context;
12 req→code = code;
13 while req→code do
14   T_wait++;
15   if req→cond ≠ TRUE then
16     if gl→server_state = DOWN then
17       if T_wait > T_w then
18         if trylock(gl→ll) then
19           if req→code then/* Request not finished */
20             req→code = NULL;
21             upgrade as server;
22           else              /* Request finished by server */
23             res = req→context;
24             upgrade as server;
25 p.end = PAPI_get_real_cyc();
26 return req→context;
```

Clients Never Wait for Too Long. Normally, a client's request will be processed very quickly, at worst after the serving time of one iteration through the request array. However, as mentioned in Section 3.3, a client may not be allowed to send requests to a remote server, and many have to wait for a local server. SANL solves this case with the dynamic server nomination scheme, which guarantees that the client or a local thread will become a server within a certain time. For client threads, SANL provides T_w as a tunable upper-bound timeout value. In our evaluation, we find that the average waiting time is only a few microseconds even under very high contention.

3.4.2 Ensuring Fairness

SANL's fairness is guaranteed in both the in-place mode and the delegation mode. In the in-place mode, the combined in-place lock is responsible to ensure fairness. Thus, we recommend combining SANL with a fair in-place lock. In the delegation mode, a server thread *iteratively* processes requests in its request array, guaranteeing that every client can be served within one iteration. Under the serving-first NUMA policy, SANL ensures global fairness via dynamic server nomination. Using Q, every thread far away from the server either enqueues itself and becomes a server or finds a local thread that can upgrade to a server within a certain time. Therefore, no client will starve in SANL.

3.4.3 Automatic Parameter Determination

Threshold values are important for SANL to appropriately apply different policies. θ_l, θ_f and L_s are related to both hardware features and the combined in-place lock. To avoid the chore of tuning them, SANL provides a profiler to automatically detect the proper values for them. For a given machine and the combined in-place lock, SANL only needs to execute the profiler once. The profiler determines θ_l, the threshold of *In-place-Delegation* mode transition, by executing a micro-benchmark with the combined in-place lock. The proper value is the average of C_l when the in-place lock and delegation lock have the same performance. θ_f, the threshold of *free-restrictive* mode transition, is determined by comparing SANL's delegation lock without NUMA-aware policy with distance-first delegation lock. The profiler sets θ_f as the value when the two locks perform equally.

L_s, the dynamic limit of the server's iteration times, dynamically changes according to the contention level C_l. SANL defines L_s as a threshold of the number of times the request density as detected is considered low. An adaptive function should be able to capture the relationship between L_s and C_l

$$L_s = f(C_l^{-1}). \tag{3}$$

We use the inverse value of C_l for simplicity to avoid fractions. The profiler computes the adaptive function f by running a micro-benchmark with different L_s values and

TABLE 3
Hardware Characteristics of Our Two Testbeds

Name	Intel Westmere-EX	AMD Interlagos
Processors	4 × Xeon E7-4850	8 × Opteron 6274
# cores	40	64
Clock rate	2.0 GHz	2.2 GHz
L1 Cache	32/32 KiB I/D	64/16 KiB I/D
L2 Cache	256 KiB	2,048 KiB
Last-level Cache	24 MiB (shared)	2 × 8 MiB (shared)
Cache Coherency Protocol	MESIF	MOESI
Interconnect	6.4 GT/s QuickPath Interconnect (QPI)	6.4 GT/s HyperTransport (HT) 3.0
Memory #Channels / #Nodes	64 GiB Sync DDR3 4 per socket / 4	128 GiB Sync DIMM 4 per socket / 8

different interval times between critical sections. During an application's execution, SANL dynamically computes C_l^{-1} with recorded cycle numbers (Algorithm 2's lines 1 and 20, and Algorithm 3's lines 1 and 25). Depending on C_l^{-1} at runtime, SANL dynamically adjusts L_s accordingly. We show details and experiments of the profiler in the evaluation section.

4 Evaluation

We evaluate SANL's performance using both micro-benchmarks and application benchmarks. The micro-benchmarks are the same as those in [16], [17]. The micro-benchmarks include 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 all tests, we vary the interval from 100 to 2,000,000 cycles. For application benchmarks, we reuse RCL's re-engineering tool to apply SANL to four popular multi-threaded applications: Berkeley DB, Memcached, Phoenix2 and SPLASH-2.

We conduct experiments on two NUMA machines. One is a 40-core machine with four 10-core Intel Xeon E7-4850 processors, running Debian 7. The other is a 64-core machine with eight 8-core AMD Opteron 6,274 processors, running Ubuntu 12.04. Their hardware characteristics are detailed in Tables 3 and 4. In both machines, we run Linux 3.14.3 + gcc 4.8.0 + glibc 2.19, with hyper-threading (HT) disabled on the Intel machine (AMD has no HT technology). We use the Pthread library to support multi-threaded

TABLE 4
Latencies (Cycles) of the Cache Coherence to Load/Store a Modified Cache Line with Difference Distances

Name	Xeon		Opteron		
Hops	same die	one hop	same die	one hop	two hops
Load	100	281	228	419	498
Store	105	302	256	463	544

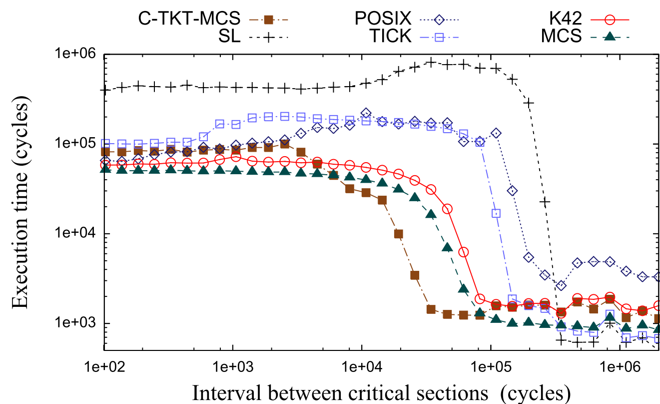

Fig. 6. Performance comparison results of six representative in-place locks.

execution. We bind each thread to a dedicated core, and thereby the thread number represents the number of testing cores. When assigning threads to cores, we adopt the common proximity-first policy: one thread will not be bound to another NUMA node until the current node is full.

4.1 In-Place Lock Selection and Parameter Determination

SANL relies on an existing lock scheme when it switches to the in-place mode. Although SANL can work with any in-place lock in principle, it is important to select the most efficient one to achieve high performance under low contention. In Fig. 6, we evaluate six representative in-place locks: naive spin lock, ticket lock, POSIX mutex, MCS lock, K42 lock and C-TKT-MCS cohort lock. To maximize C-TKT-MCS's performance, we adopt the "may-pass-local" method with the cohorting-threshold set as "infinite". MCS and C-TKT-MCS locks are the best ones under various contention levels. Although C-TKT-MCS lock performs better than MCS under medium contention (e.g., when the interval is around 10,000 cycles), when the contention is low (e.g., the interval $>$ 100,000 cycles), MCS is 1.25 times faster than C-TKT-MCS lock. Therefore, we select MCS to work with SANL.

To determine the proper threshold values for θ_l, L_s and θ_f (as shown in Fig. 2), which are all related to the CPU's architecture, SANL's profiler is executed in three steps: 1) For θ_l, the threshold of *In-place-Delegation* mode transition, the profiler runs the micro-benchmark with the selected in-place lock and with SANL without NUMA-aware policy (represented as non-NUMA); θ_l is set as the average of C_l when the delegation lock just outperforms the in-place lock; in our evaluation, θ_l is 17 percent for the Intel machine and 6 percent for the AMD machine. 2) To determine L_s, the dynamic limit of the server's iteration times, the profiler runs non-NUMA-aware SANL with L_s varying from 0 to 1,000. According to the results, we find that L_s is highly related to C_l^{-1}, which is shown in Fig. 7 with secondary polynomial fitting. Thus, L_s can be derived from C_l using the fitting function f at runtime. 3) To determine θ_f, the threshold of *free-restrictive* mode transition, the profiler compares the non-NUMA-aware policy with the distance-first NUMA policy, and θ_f is set as the average of C_l when the two policies perform equally; in our evaluation, θ_f is 56 percent for the Intel machine and 40 percent for the AMD machine. The exact results on the two platforms are

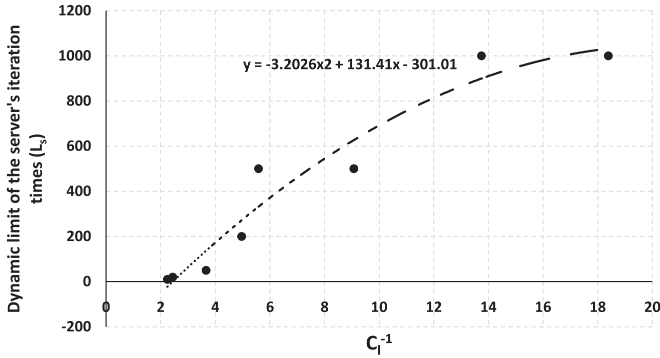

Fig. 7. The relation between L_s and C_l^{-1} with a secondary polynomial fitting function *f*.

TABLE 5
The Profiled Values of θ_l and θ_f in Our Two Testbeds

	θ_l	θ_f
Intel	17%	56%

	θ_l	θ_f
AMD	6%	40%

shown in Table 5. In both machines, the profiler takes an average of 45 minutes. In Fig. 8, we show the performance results of the non-NUMA-aware, distance-first and serving-first delegation lock schemes with the interval varying from 100 to 25,000 cycles. It can be seen that under very high contention (e.g., 100 cycles to 3,000 cycles), distance-first policy and serving-first policy perform similarly; but when the node-level contention slackens off, the efficiency of distance-first policy starts to decline significantly, becoming even worse than non-NUMA-aware policy. The reason is that when there are only a few requests in the local node, the server frequently becomes idle and is re-nominated in another NUMA node, generating excessive cache misses.

To determine the proper default value for T_s, the static limit of the server's iteration times, and T_w, the maximum spinning times of client threads, we conduct a micro-benchmark on the AMD machine with varied T_s and T_w respectively, shown in Fig. 9. The longer a server thread serves, the higher data locality SANL achieves. Specifically, when T_s is small (e.g., 1 or 10), SANL cannot make good use of the delegation mode, because a server thread conducts limited requests from clients and it downgrades. In contrast, when T_s is larger than 100, a server thread is highly utilized. Fig. 10 shows the results with varied T_w. T_w is a tunable upper-bound timeout value for clients and determines the response time of threads. We can see that a small value for T_w causes high contention for upgrading to a server thread, while a large value has little influence on the performance of SANL. Based on Figs. 9 and 10, we set T_s to 100 and T_w to 10,000 as the default values. In our evaluation, we find that the average serving time is less than one millisecond, and the average waiting time is only a few microseconds even under very high contention.

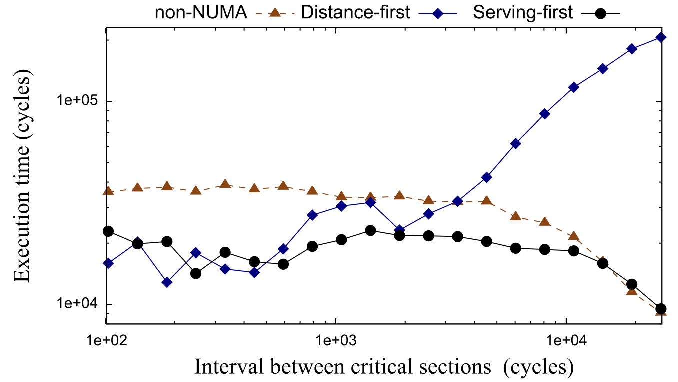

Fig. 8. Performance comparison results of non-NUMA-aware, distance-first and serving-first delegation lock schemes.

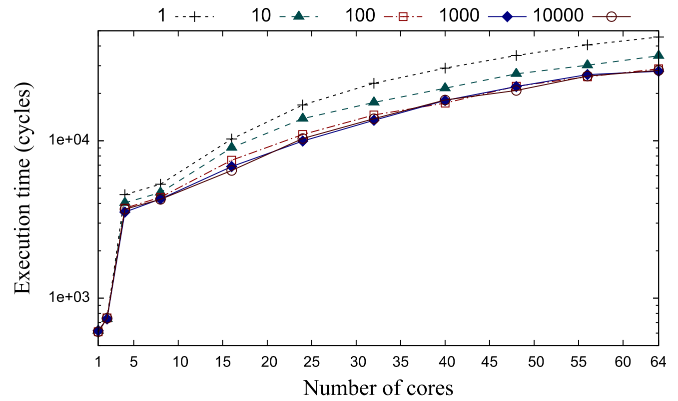

Fig. 9. Execution times with varied T_s under different thread numbers on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 1 and the interval is 103 cycles.

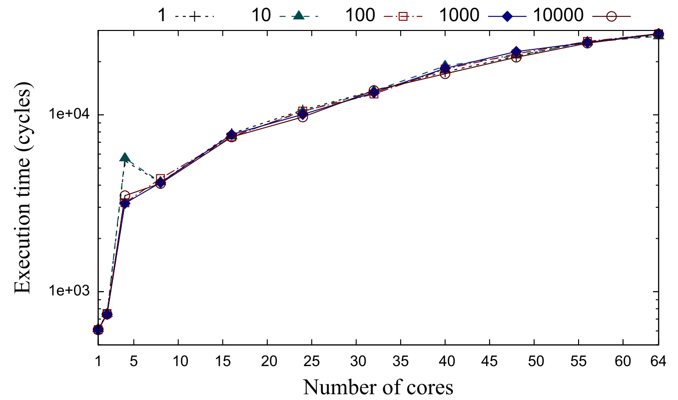

Fig. 10. Execution times with varied T_w under different thread numbers on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 1 and the interval is 103 cycles.

4.2 Micro-Benchmark

We evaluate both non-NUMA-aware SANL (SANL) and NUMA-aware SANL (NUMA-SANL) with seven other representative locks: SL, POSIX mutex, MCS, C-TKT-MCS cohort lock, Flat Combining, H-Synch and RCL. Since RCL requires a static core to act as the server, the number of client threads is *N-1*. In the other locks (including SANL), *N-1* threads run simultaneously to contend for the critical section. Thus, the micro-benchmark measures the execution time of *N-1* client threads and a server thread for RCL, and *N-1* threads for the other locks. On the Intel machine, N is 40. For the AMD machine, N is 64. The configuration confirms RCL and other

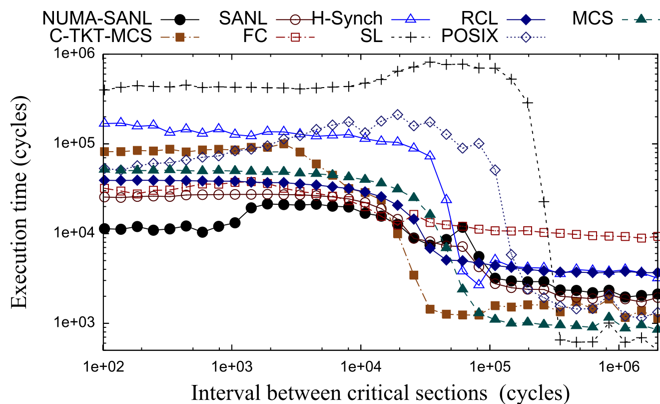

Fig. 11. Execution times under different contention levels on the Intel machine (four NUMA nodes). The number of shared cache lines in the critical section is 1.

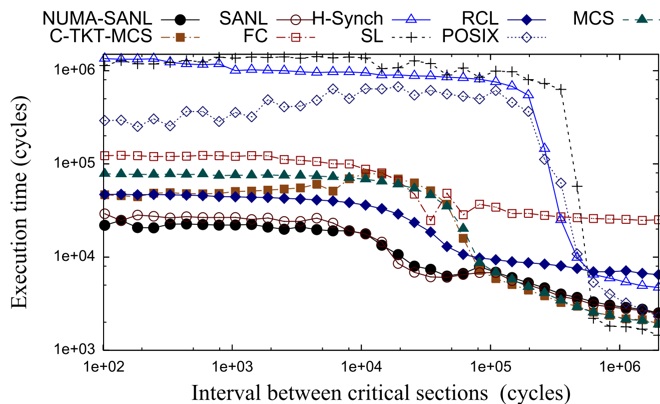

Fig. 13. Execution times under different contention levels on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 1.

locks have the same number of clients. Besides, RCL has one more core for the server. RCL actually utilizes more hardware resources. In the experiments, we also measure the effect of data locality by varying the number of shared cache lines from 1 to 5 when accessing the critical section.

Figs. 11 and 12 show the results of the Intel machine with four NUMA nodes. Taking Fig. 11 as an example, SANL's performance changes in three phases: 1) When the contention is high (e.g., when the interval is smaller than 30,000 cycles), in-place locks (SL, POSIX mutex, etc.) scale very poorly: at one or two orders slower than SANL. Even compared with C-TKT-MCS lock and MCS lock, the two best in-place locks, non-NUMA SANL is 3.2 and 2.0 times faster; and the speedup of our NUMA-SANL is much larger: 7.2 and 4.6 times respectively. Though H-Synch is also a NUMA-aware delegation lock, its performance is even lower than POSIX mutex and FC. 2) When the contention is mild (e.g., between 30,000 and 100,000 cycles), SANL shows some limitations, probably because SANL has the frequent server nomination in delegation mode. 3) When the contention is low (e.g., when the interval is larger than 100,000 cycles), SANL is slower than MCS lock (1,573 cycles versus 803 cycles). For RCL, the execution time is 3,642 cycles. From the experimental results of the following application benchmarks, we conclude SANL's overhead under low contention should be acceptable for most contended lock instances in multi-thread applications. The results with five shared cache lines, as shown in Fig. 12, are similar. The results indicate that SANL has good data locality of shared cache lines as the data in delegation mode mostly remains on the server thread.

On the AMD machine with eight NUMA nodes, Figs. 13 and 14 show that SANL's advantage is more apparent: it outperforms the others in almost all contention levels. Under high contentions, compared with in-place locks, SANL is 14 times faster than POSIX mutex, 2.4 times faster than C-TKT-MCS lock and 3.8 times faster than MCS lock. When compared with delegation locks, SANL is around 5.9 times and 2.3 times faster than FC and RCL respectively. Under low contentions, SANL performs very closely to MCS (2,520 cycle versus 2,368 cycles). The results show that SANL is very suitable for large NUMA systems. Through evaluating the number of L2 cache misses, we find SANL in both machines show similar cache miss improvement than other locks. Because the AMD machine has longer latencies of cache coherence to load/store a modified cache line than the Intel one as shown in Table 4, the latency benefits of SANL are higher on AMD than on Intel.

To investigate SANL's limitation when the contention is mild, we analyze the switch times between in-place mode and delegation mode, and the server nomination times in delegation mode. For the switch times, the results show that SANL is stable in either in-place mode or delegation mode and cases like Stage 2 in Fig. 3 happen less than 2 times.

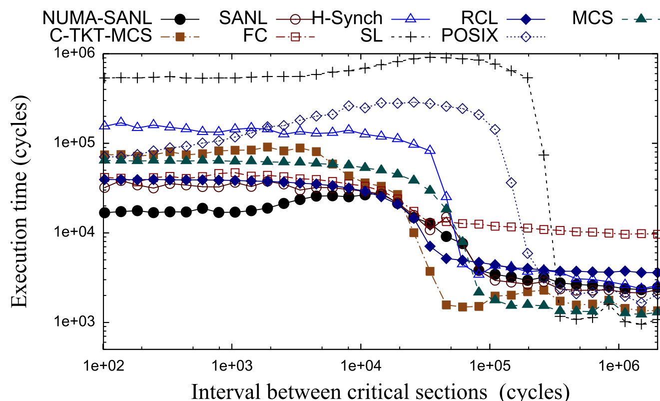

Fig. 12. Execution times under different contention levels on the Intel machine (four NUMA nodes). The number of shared cache lines in the critical section is 5.

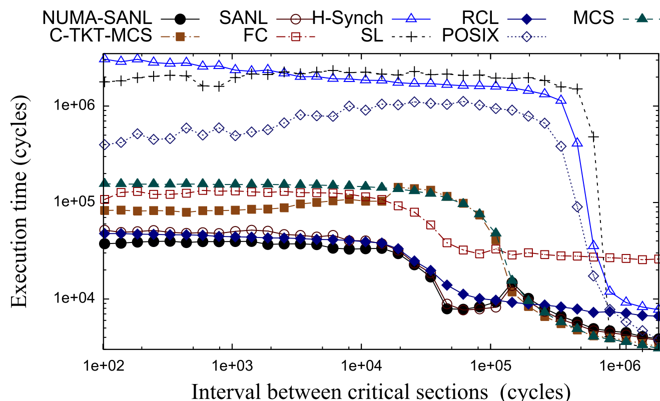

Fig. 14. Execution times under different contention levels on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 5.

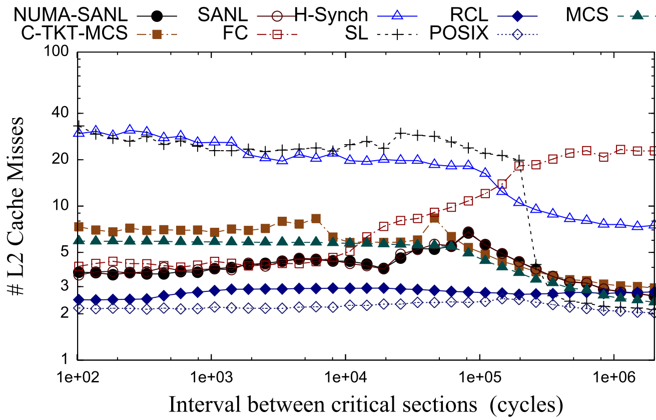

Fig. 15. The number of L2 cache misses on the Intel machine, with one shared cache line in the critical section.

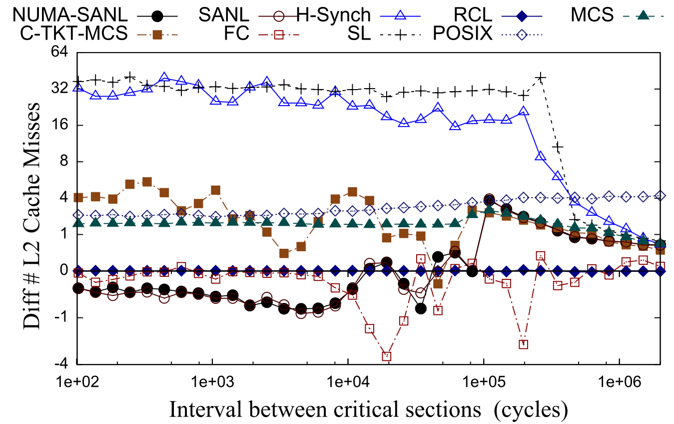

Fig. 17. Difference between the number of L2 cache misses on Intel machine between 5 and 1 shared cache lines in the critical section.

This confirms the effectiveness of our voting algorithm (Algorithm 1). For the server nomination times, Fig. 19 shows the results on the AMD machine: when the interval is between 30,000 and 100,000 cycles, the number of server nomination increases rapidly. This illustrates that the contention is higher than θ_l and SANL turns to delegation mode. However, when the contention is not high enough and the server is not fully utilized by requests from clients, frequent server nominations decrease the data locality in delegation mode and bring forth more cache misses. This explains the reason why SANL's performance decreases and approaches Flat Combining's. Both delegation mode and in-place mode are not efficient for mild contention levels (e.g., the interval between 30,000 and 100,000 cycles). It is our future work to address this problem to achieve seamless adaptation between delegation mode and in-place mode.

We also evaluate the overhead of SANL's profiling scheme, voting scheme and the Id manager. Under both high and low contentions when accessing 1 shared cache line, the profiling scheme takes around 100 cycles, calling *PAPI* interfaces to get time information; the voting scheme takes around 217 cycles. Even under high contention, the overheads only account for 0.39 and 0.84 percent of the critical section's execution time respectively. We attribute the efficiency of the voting scheme to the boundary optimization in Algorithm 1. For the *Id manager*, we evaluate it with a micro-benchmark in which 64 threads repeatedly require Id and release Id. We find that under high contention, a pair of the *Id manager's* acquire and release operations takes 3518.51 cycles; under low contention, it takes only 102 cycles.

In Figs. 15 and 16, we show the effect of data locality based on the number of L2 cache misses. Fig. 17 shows the difference of L2 cache misses when the shared cached line increases from 1 to 5. The results of C-TKT-MCS lock, MCS lock and POSIX mutex show that in-place locks cause higher caches misses because of either lock request contention or shared data contention. In Fig. 17, their cache miss differences are all above 0. RCL incurs very few cache misses because its server thread never changes the core, maintaining good data locality. Like RCL, FC also has a low cache miss rate at high contention, because most of the time the critical section can stay with the same server; however, under low contention, the number of cache misses increases because the server frequently downgrades when there are not many requests to process. With SANL, regardless of the size of the shared data, it maintains a reasonable number of cache misses in almost all scenarios; when the critical section becomes longer (Fig. 16), SANL's cache miss is very similar to RCL because it behaves much like a static delegation lock. In Fig. 17, under high contention, the number of cache misses of SANL even decreases due to increased server utilization and fewer server re-nominations. With low contention, SANL adapts to in-place lock and so it brings a similar number of cache

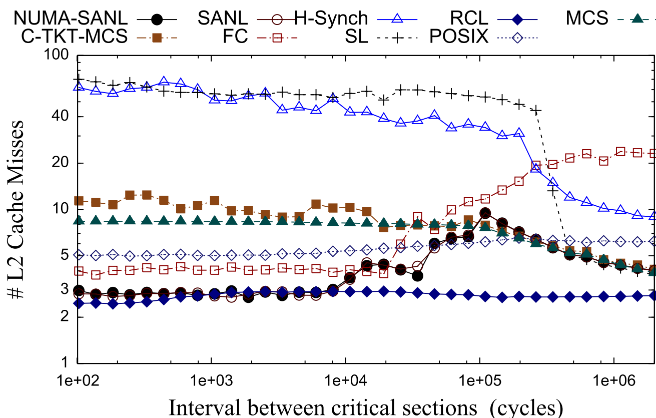

Fig. 16. The number of L2 cache misses on the Intel machine, with five shared cache lines in the critical section.

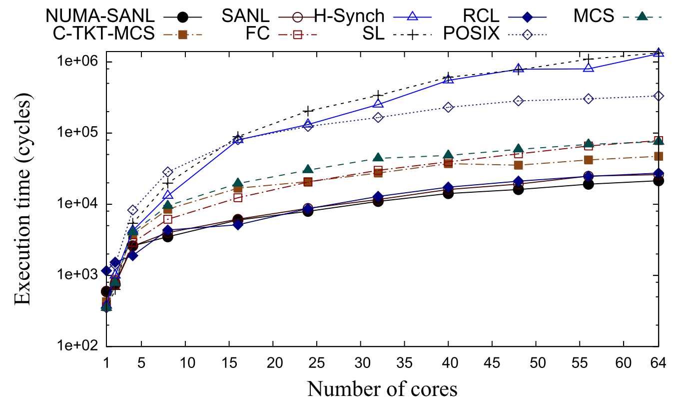

Fig. 18. Execution times under different thread numbers on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 1 and the interval is 103 cycles.

TABLE 6
Profiling Results of the Evaluated Applications on Our 40-Core Intel Machine

Application		Lock usage with 40 cores		
		Description	# locks	% in CS
Berkeley DB with TPC-C	Order Status	DB struct. access	11	59.1%
	Stock Level	DB struct. access	11	76.2%
Memcached†	Get	Hashtable access	1	84.7%
	Set	Hashtable access	1	61.6%
Phoenix 2	Linear Regression	Task queue access	1	88.7%
	String Match	Task queue access	1	69.6%
	Matrix Multiply	Task queue access	1	65.9%
SPLASH-2	Radiosity	Linked list access	1	56.5%
	Raytrace/Balls4	Counter increment	1	85.1%
	Raytrace/Car	Counter increment	1	91.3%

† *18 cores are used for the server and the other 18 cores are used for clients.*

misses to MCS. On the AMD machine, Fig. 18 shows the execution time of varied client threads. When the number of threads is small, `SANL` performs very closely to MCS (593 cycle versus 350 cycles). When the number of threads is large, `SANL` is more than 3.5 times faster than in-place locks, and 1.3 times faster than delegation locks. However, when the number of threads is 4, `SANL` is 1.37 times slower than RCL because of the frequent server nominations of dynamic delegation lock. From the results of the micro-benchmarks, we conclude that `SANL` outperforms its counterparts under high contention and has moderate overhead in low contention situations.

For memory footprint, `SANL` is similar to RCL. The main usage of memory is the request array. On the Intel machine with 40 cores, the memory footprint is around 1.25 MiB; on the AMD machine with 64 cores, the memory footprint is around 2 MiB.

4.3 Application Performance

To investigate `SANL`'s performance in real applications, we apply it to four popular multi-threaded applications: BerkeleyDB, Memcached, Phoenix2 suite and SPLASH-2 suite. We select the benchmarks with high lock contention in Phoenix2 suite and SPLASH-2 suite. In our 40-core Intel machine, Table 6 shows the profiling results of the percentage of time they spend executing the critical section with their default POSIX mutex lock.[2] In all applications, lock contending time occupies more than half of the execution time. In Memcached, the `GET` operation spends nearly 90 percent of the time in the critical section. We configure RCL and other locks with the same number of clients for all applications. In Phoenix2, SPLASH-2 and Memcached, since there is only one lock, RCL is configured with *N+1* cores (one server plus *N* clients) while the other locks are configured with *N* cores (*N* clients); in Berkeley DB which contains 11 locks, RCL is configured with *N+2* cores, reserving two cores for server threads to process the requests and leaving the remaining *N* cores for the clients. Because the applications have more servers for RCL, RCL utilizes more hardware resources than other locks in our application evaluations.

2. Similar results are also reported in [16], [17] with a 48-core machine.

4.3.1 Berkeley DB

We record the performance of two TPC-C transaction types: `StockLevel` and `OrderStatus`. In the tests of `StockLevel` on the Intel machine, Fig. 20 shows that `SANL` delivers the highest throughput in almost all tests. Among the other seven selected locks, RCL and C-TKT-MCS lock perform the best. Compared to RCL, `SANL` achieves at least 1.64 times higher throughput when the number of clients is smaller than 5 (low contention), because it combines with MCS lock in the in-place mode; when there are 10 clients (medium contention), `SANL` is still 10 percent better than RCL; when the number of clients further increases (high contention), `SANL` outperforms RCL by 24 to 49 percent. C-TKT-MCS lock performs similarly with `SANL` when the number of clients is smaller than 20; but `SANL` shows 15-25 percent higher throughput when there are more than 25 clients. The results on the AMD machine (Fig. 21) also show that `SANL` outperforms the other locks in most settings.

With `OrderStatus` transactions, `SANL` performs at least 16 percent better than the other locks on the Intel machine, and achieves up to 66 percent throughput improvement on the AMD machine (the comparison results are very similar to Figs. 20 and 21, so we do not display them here). Unlike RCL which reserves cores for servers and may introduce the false serialization problem, `SANL` dynamically

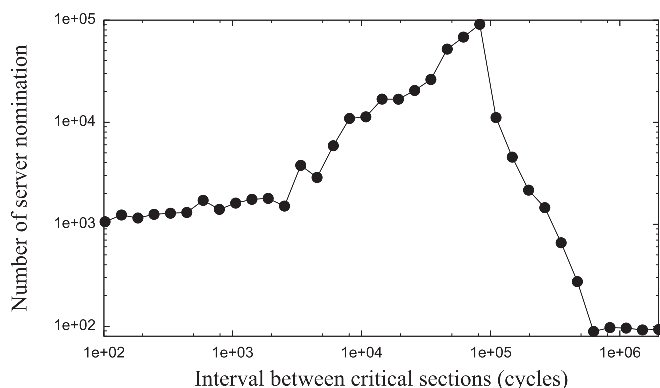

Fig. 19. Number of server nomination under different contention levels on the AMD machine (eight NUMA nodes). The number of shared cache lines in the critical section is 1.

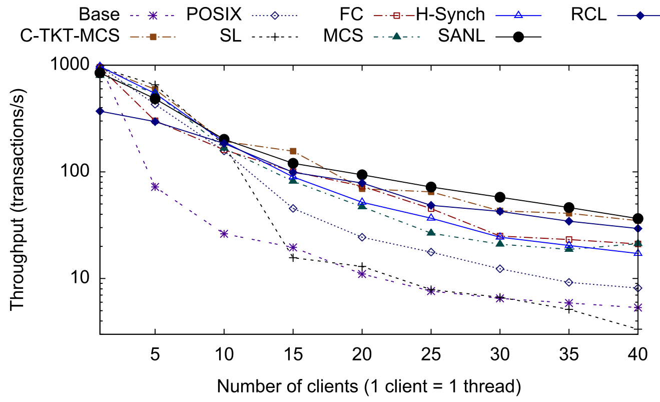

Fig. 20. Throughput results of BerkelyDB's `StockLevel` transaction on the Intel machine.

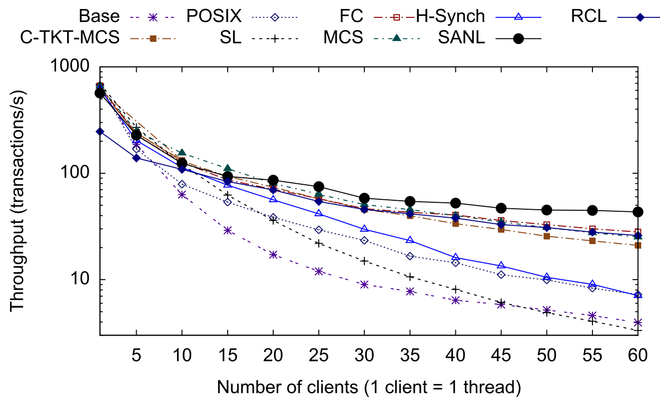

Fig. 21. Throughput results of BerkelyDB's `StockLevel` transaction on the AMD machine.

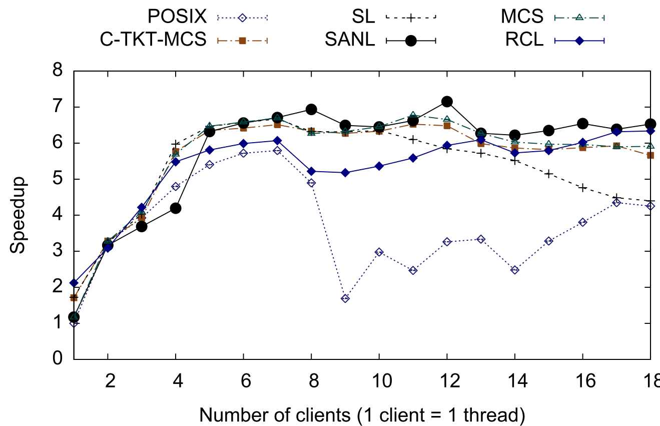

Fig. 22. Speedup of Memcached's `SET` on the Intel machine. Single-thread POSIX as baseline.

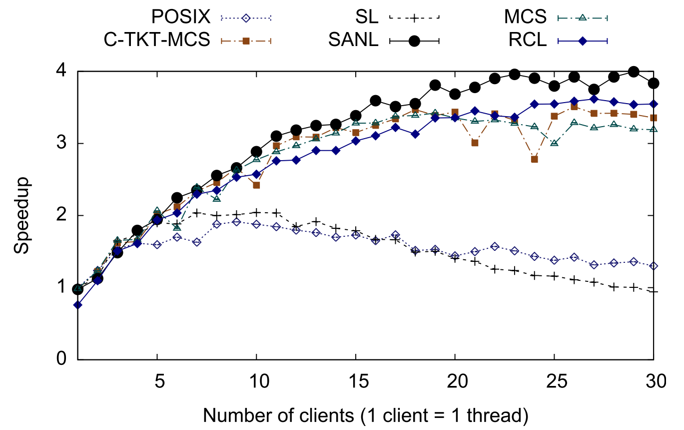

Fig. 23. Speedup of Memcached's `SET` on the AMD machine. Single-thread POSIX as baseline.

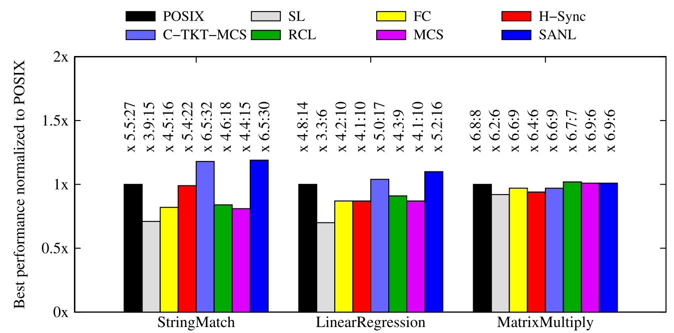

Fig. 24. Phoenix results on the Intel machine: Best performance of each lock relative to POSIX mutex.

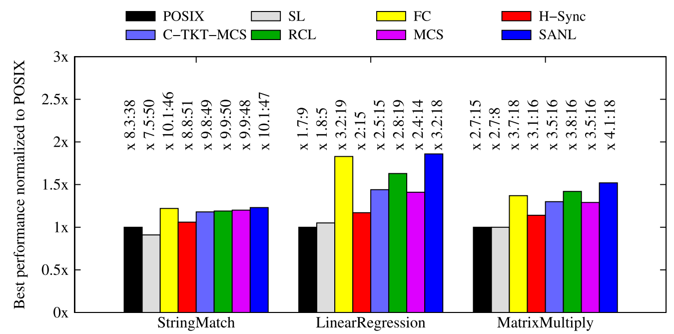

Fig. 25. Phoenix results on the AMD machine: Best performance of each lock relative to POSIX mutex.

nominates server threads for different lock instances. Since all server threads can process lock requests at the same time, `SANL` can more efficiently handle nested critical sections. In addition, `SANL`'s server thread can downgrade at an appropriate time to save core resources. For in-place locks, the throughput decreases considerably when the contention becomes high.

4.3.2 *Memcached*

In Memcached, we evaluate pure `GET` and pure `SET` requests. To avoid the bottleneck from the network, we put clients and servers on the same machine, with an evenly allocated number of cores. The results with Flat Combining and H-Synch are omitted because Memcached requires periodically blocking on condition variables, which is not supported in FC and H-Synch. When testing the `GET` operation, we find that all locks perform similarly (so the figures are not displayed here). A possible explanation is that its critical sections are long and thus acquiring and releasing locks is less likely to become a bottleneck than in other applications. The results of `SET` operation with the Intel and AMD machine are shown in Figs. 22 and 23 respectively. On both platforms, `SANL` consistently outperforms the others in almost all contention levels. For example, in Fig. 22, `SANL` performs better than MCS and RCL by 6 to 33 percent respectively. `SANL` only experiences performance regression when there are four threads, which is caused by

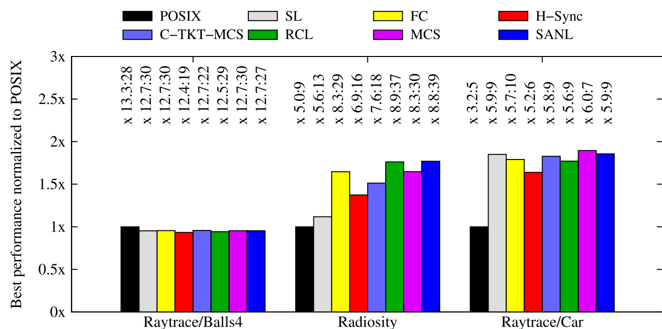

Fig. 26. SPLASH-2 results on the Intel machine: Best performance of each lock relative to POSIX mutex.

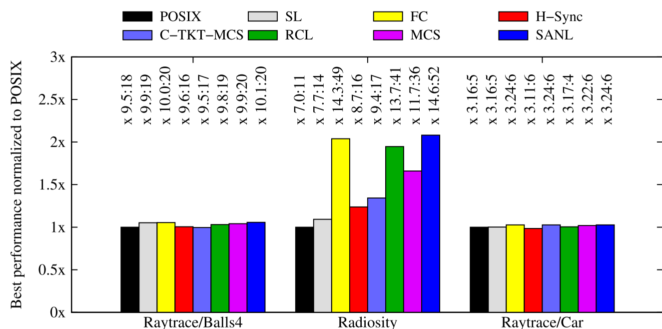

Fig. 27. SPLASH-2 results on the AMD machine: Best performance of each lock relative to POSIX mutex.

the overhead of frequent server nominations of dynamic delegation locks.

4.3.3 Phoenix2

The results of Phoenix2 tests are shown in Figs. 24 and 25. At the top of each figure, ($\times\ \alpha : n$) reports the maximum speedup of α over the single-thread execution time, achieved with n threads/cores. The results are normalized to POSIX mutex's performance. In the `StringMatch` tests, `SANL` has similar performance to MCS lock when the number of cores is small and has similar performance to RCL and FC when the number of cores increases. With `LinearRegression`, `SANL` shows improvement over other locks by up to 58 percent. In the `MatrixMultiply` tests, the advantage of `SANL` over the others is marginal on the Intel machine, but on the AMD machine, `SANL` achieves more than 50 percent performance improvement.

4.3.4 SPLASH-2

SPLASH-2 is a benchmark suite to evaluate parallel scientific applications for cache-coherent architectures. The results with `SANL` are shown in Figs. 26 and 27. In the `Raytrace/Balls4` and `Raytrace/car` tests, the maximum speedup of tested locks are similar. In the `Radiosity` tests, `SANL` performs 7-25 percent faster than MCS lock. In general, `SANL` consistently performs as the best one among all lock schemes in SPLASH-2.

5 Conclusion and Future Work

This paper describes `SANL`, a scalable adaptive NUMA-aware locking mechanism that combines the advantages of in-place lock for low contention and delegation lock for high contention. `SANL` uses a voting algorithm to profitably switch between the two lock schemes during runtime. In addition, we design and implement a more scalable NUMA-aware policy to work with `SANL` in its delegation-lock mode. Evaluations based on a set of multi-threaded applications show that `SANL` outperforms state-of-the-art lock algorithms under both low and high contentions.

In future, we plan to focus on the mitigation of the situations in which there is frequent switching between locking schemes. We could also enhance the re-engineering tool we used in the evaluation to support codes with thrown exceptions, non-lexically scoped locking, etc. Furthermore, as power management becomes increasingly important, it would be meaningful to investigate the tradeoff between a lock's performance and power consumption with finer-grained power tuning knobs.

References

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

[2] Z. Radovic and E. Hagersten, "Hierarchical backoff locks for nonuniform communication architectures," in *Proc. 9th Int. Symp. High-Performance Comput. Archit.*, 2003, Art. no. 241.

[3] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards, "Simple and fast biased locks," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 65–74.

[4] Y. Cui, et al., "Reducing scalability collapse via requester-based locking on multicore systems," in *Proc. IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2012, pp. 298–307.

[5] J. L. Abellán, J. Fernández, and M. E. Acacio, "GLocks: Efficient support for highly-contended locks in many-core CMPs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 893–905.

[6] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[7] P. E. McKenney, et al., "Read-copy update," in *Proc. Linux Symp.*, 2001, pp. 175–184.

[8] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 199–210.

[9] V. Luchangco, D. Nussbaum, and N. Shavit, "A hierarchical CLH queue lock," in *Proc. 12th Int. Conf. Parallel Process.*, 2006, pp. 801–810.

[10] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 247–256.

[11] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 33–48.

[12] Y. Oyama, K. Taura, and A. Yonezawa, "Executing parallel programs with synchronization bottlenecks efficiently," in *Proc. Int. Workshop Parallel Distrib. Comput. Symbolic Irregular Appl.*, 1999, pp. 182–204.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. 22nd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 355–364.

[14] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 325–334.

[15] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 257–266.

[16] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 6–6.

[17] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Fast and portable locking for multicore architectures," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, pp. 13:1–13:62, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2845079

[18] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 33–48.

[19] B. Fitzpatrick, "Memcached: A distributed memory object caching system," 2011. [Online]. Available: http://memcached.org/

[20] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1999, pp. 43–43.

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Performance Comput. Archit.*, 2007, pp. 13–24.

[22] U. of Delaware, "The modified splash-2 home page," 2007. [Online]. Available: http://www.capsl.udel.edu/splash

[23] J. P. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *Stanford Univ., Stanford, CA,* USA, Tech. Rep. CSL-TR-91–469, 1992.

[24] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Proc. 16th Annu. Int. Symp. Comput. Archit.*, 1989, pp. 396–406.

[25] M. Auslander, D. Edelsohn, O. Krieger, B. Rosenburg, and R. Wisniewski, "Enhancement to the MCS lock for increased functionality and improved programmability," U.S. Pantent Appl. US20030200457 A1, 2002.

[26] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "Readers" and "Writers"," *Commun. ACM*, vol. 14, no. 10, pp. 667–668, Oct. 1971.

[27] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive reader-writer locks," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2014, pp. 219–230.

[28] Y. Lev, V. Luchangco, and M. Olszewski, "Scalable reader-writer locks," in *Proc. 21st Annu. Symp. Parallelism Algorithms Archit.*, 2009, pp. 101–110.

[29] S. S. Bhat, "percpu_rwlock: Implement the core design of Per-CPU Reader-Writer Locks," 2013. [Online]. Available: https://patchwork.kernel.org/patch/2157401

[30] M. Arbel and A. Morrison, "Predicate RCU: An RCU for scalable concurrent updates," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 21–30.

[31] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, "NUMA-aware reader-writer locks," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 157–166.

[32] T. Craig, "Building FIFO and priority queuing spin locks from atomic swap," *Univ. Washington, Seattle, WA, USA,* Tech. Rep. 93–02-02, Feb., 1993.

[33] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High performance locks for multi-level NUMA systems," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 215–226.

[34] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, locality-preserving locks," in *Proc. 21st ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, Art. no. 22.

[35] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, 2009, pp. 3–14.

[36] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, "Adaptive integration of hardware and software lock elision techniques," in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2014, pp. 188–197.

[37] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "CPHASH: A cache-partitioned hash table," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 319–320.

[38] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA locks," in *Proc. 23rd ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 65–74.

[39] D. Petrović, T. Ropars, and A. Schiper, "On the performance of delegation over cache-coherent shared memory," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2015, Art. no. 17.

[40] I. Calciu, et al., "Message passing or shared memory: Evaluating the delegation abstraction for multicores," in *Proc. 17th Int. Conf. Principles Distrib. Syst.*, 2013, pp. 83–97.

[41] D. Klaftenegger, K. Sagonas, and K. Winblad, "Delegation locking libraries for improved performance of multithreaded programs," in *Proc. Euro-Par*, 2014, pp. 572–583.

Mingzhe Zhang received the BEng degree (rank 1st out of 82) in software engineering from Fudan University, Shanghai, China, in 2013. He is currently working toward the PhD degree in computer science at the University of Hong Kong. His research interests include mainly in multi-core synchronization technologies and fault tolerance technologies, including highly scalable lock algorithms, and fault tolerance techniques based on non-volatile memory. He received the China National Scholarship in 2011 and the Google Excellence Scholarship in 2012.

Haibo Chen received the BEng and PhD degrees in computer science from Fudan University. He is a professor in the School of Software, Shanghai Jiao Tong University. His research interests include improving the performance and dependability of computer systems. He has published more than 50 papers in various reputable conference proceedings, such as SOSP, USENIX ATC, EuroSys, PPoPP, HPCA, ISCA, etc. He is a senior member of the IEEE and the ACM.

Luwei Cheng received the PhD degree in computer science from the University of Hong Kong, in 2015. He is currently a research scientist with Facebook's Data Infrastructure Team. His research interests include mainly in performance-related things of cloud data centers, including operating systems, virtual machines, datacenter networking, and distributed computing/storage frameworks. He received the Best Student Paper Award in IEEE/ACM UCC in 2011, the Hong Kong PhD Fellowship Award in 2012, the Microsoft Research Asia Fellowship Award in 2013, and Best Paper Award in APSys in 2016.

Francis C. M. Lau received the PhD degree in computer science from the University of Waterloo, Waterloo, Canada, in 1986. He has been a faculty member in the Department of Computer Science, University of Hong Kong, Hong Kong, since 1987, where he served as the department chair from 2000 to 2005 and is now an associate dean of the Faculty of Engineering. He was an honorary chair professor in the Institute of Theoretical Computer Science, Tsinghua University, Beijing, China, from 2007 to 2010, and is currently a visiting chair professor with Jinan University, Guangzhou, China. He is the editor-in-chief of the *Journal of Interconnection Networks*. His research interests include computer systems and networking, algorithms, HCI, and application of IT to arts. He is a senior member of the IEEE

Cho-Li Wang recevied the BS degree in computer science and information engineering from National Taiwan University, in 1985 and the PhD degree in computer engineering from the University of Southern California, in 1995. He is currently a professor in the Department of Computer Science, University of Hong Kong. His research is broadly in the areas of parallel architecture, software systems for cluster computing, and virtualization techniques for cloud computing. His recent research projects involve the development of parallel software systems for multicore/GPU computing and multi-kernel operating systems for future many-core processor. He has published more than 150 papers in various peer reviewed journals and conference proceedings. He is/was on the editorial boards of several scholarly journals, including the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Computers*, and the *Journal of Information Science and Engineering*. He also serves as a coordinator (China) of the *IEEE Technical Committee on Parallel Processing*. He is a senior member of the IEEE

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.