TR: A Lightweight Log-based Deferred Update for Linux Kernel Scalability

***Abstract*—In highly parallel computing systems with many-cores, a few critical factors cause performance bottlenecks severely limiting scalability. The kernel data structures with high update rate naturally cause performance bottlenecks due to very frequent locking of the data structures. There have been research on log-based synchronizations with time-stamps that have achieved significant level of performance and scalability improvements. However, sequential merging operations of the logs with time-stamps pose another sources of scalability degradation.**

To overcome the scalability degradation problem, we introduce a lightweight log-based deferred update method, combining the log-based concepts in the distributed systems and the minimal hardware-based synchronization in the shared memory systems. The main contributions of the proposed method are:(1) we propose a lightweight log-based deferred update method, which can eliminate synchronized time-stamp counters that limits the performance scalability;and (2) we implemented the proposed method in the Linux 4.5-rc6 kernel for two representative data structures (anonymous reverse mapping and file mapping) and evaluated the performance improvement due to our proposed novel light weight update method. Our evaluation study showed that application of our method could achieve from 1.5x through 2.7x performance improvements in 120 core systems.

***Keywords*-operating systems; Linux kernel; many-core; scalability; update-heavy**

I. Introduction

Achieving performance scalability has been a most important factor in highly parallel systems with many cores (e.g., over 100 cores). The performance scalability of a whole system is naturally limited by scalability of underlying operating system kernel; Linux kernel has been widely considered in ordinary systems. Previous research revealed that Linux kernel has significant problems limiting performance scalability in many core systems [1] [2] and the major sources of the problems are update lock contention in a few kernel data structures [3] [4].

Early research accomplishments regarding the update serialization problems include a number of concurrent update methods [5] [4] [6]. Such research provides bases to solve the update serialization problems, but does not effectively handle serious scalability bottleneck for update-heavy data structures. Log-based algorithms [7] [8] have been proposed to solve this update serialization problem by reducing cache coherence-related overheads for update-heavy data structures. When update operations occur, log-based algorithm logs the update operation and applies all operation logs to the data structure before read operation, so readers can read up to date data structure in a way similar to CoW(Copy On Write) [9] [10].

Among the log-based methods, S. Boyd-Wickizer *et al.* proposed OpLog [8], where each update operation generates a log with synchronized time-stamp counters and serialization of the logs based on the time-stamps solves the scalability bottleneck for update-heavy data structures: the loggings are performed on to per-core memory instead of shared memory and thus eliminates cache communication overhead. However, the OpLog still has problems in scalability since the synchronized time-stamp counters necessitates time-stamp merging and ordering processes leading to performance scalability problems in high CPU core counts.

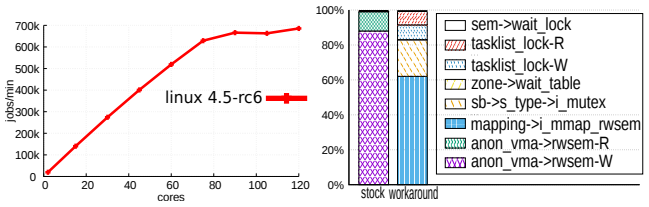
We propose a novel lightweight log-based differed update method(LDU) to achieve the maximum performance scalability for update-heavy data structures solving the problems of sequential processing raised from the previous research: cache communication overhead in logging and time-stamp management cost during log ordering. Such improved scalability could be achieved through combination of widely known log-based concurrent update concept and our own way of efficient implementation methods of the log management scheme: log record slot reuse in the log queue and using minimal hardware-based synchronization method(compare and swap, test and set, atomic swap).

To evaluate our approach, we applied the LDU to Linux kernel reverse page mappings(anonymous page mapping, file page mapping) that are considered as the major sources of limiting performance scalability due to their update-heavy characteristics. We implemented the LDU in a Linux 4.5-rc6. We evaluated the performance and scalability using a fork-intensive workload- AIM7 [11], Exim [12] from MOSBENCH [13] and Lmbench [14]-our design improves throughput and execution time on 120 core by 1.5x, 2.6x, 2.7x respectively, relative to stock Linux.

Contributions. Our research makes the following contributions:

- We have developed a novel lightweight log-based deferred update method eliminating the sources of limiting performance scalability in update-heavy data structures with efficient log management implementation.
- We applied the LDU in Linux kernel to two reverse mapping(anonymous, file) on an 120 core system to reduce fork scalability bottleneck. Our design improved throughput and execution time from 1.5x through 2.7x on 120 core.

The rest of this paper is organized as follows. Section II



(a) AIM7-multiuser scalability (b) Lock wait time on 120 core

Figure 1: Scalability of AIM7 multiuser and wait time to acquire locks on 120 core. This workload simultaneously create many processes. The (a) shows up to 75 core, the stock Linux scales linearly, then it flattens out. The (b) left bar represents anonymous reader-writer semaphore causes a scalability bottleneck, and The (b) right bar shows file mapping reader-writer semaphore causes a scalability bottleneck.

describes the background and the Linux scalability problem. Section III describes the design of the LDU algorithm and Section IV explains how to apply the LDU to Linux kernel, and Section V shows the results of the experimental evaluation. Section VI describes related work with our research. Finally, section VII concludes the paper.

II. Background and Problem

Applications' performance would be limited by the operating system kernel when the operating system kernel does not scale well [15] [16]. To analyze the current status of the operating system scalability, we measured the performance trends of the AIM7-multiuser benchmark on Linux with various CPU cores. The AIM benchmark has, until recently, been widely used in research area and the Linux community [17] [18]. The AIM7-multiuser workload simultaneously creates many processes with disk-file operations, virtual memory operations, pipe I/O, and arithmetic operations. The result of figure 1(a) shows that Linux shows significantly limited performance scalability when CPU core exceeds 75.

To understand the sources of scalability bottleneck on 120 core systems, we profiled a lock contention using the `lock_stat` [19], a Linux kernel lock profiler that reports how long each lock is held and the wait time to acquire the lock. The figure 1(b) shows the cost of acquiring the lock for the AIM7-multiuser running on a 120 core system. Results of the `lock_stat` that the main source of lock contention was the anonymous reverse mapping semaphore(`anon_vma->rwsem`), which was caused by simultaneously creating a number of processes. To understand further bottlenecks, we intentionally removed the kernel codes causing anonymous reverse mapping which has been revealed as the most significant source of the problem. When the anonymous reverse mapping code was removed, the second major source of the bottleneck was file reverse mapping reader-writer semaphore(`mapping->i_mmap_rwsem`).

Our background study showed that both the anonymous reverse mapping reported from the Linux community [20] and the file reverse mapping reported from S. Boyd-Wickizer [8] are significant factors in fork scalability problem. Thus, in order to perfect scalability of the fork, both the file reverse mapping and the anonymous reverse mapping should be executed concurrently without any lock.

Existing research accomplishments to achieve scalable concurrent update in many core systems are categorized into two methods: non-blocking algorithms [21] [22] [23] in early stages and log-based algorithms. In non-blocking algorithms, update operation observes against the current value in global data structure, and they execute an atomic compare and swap(CAS) to compare the against value. When the value has been overridden, the updater must be retried. Consequently, both the repeated global CAS operation and the iteration loop caused by CAS fails will result in bottlenecks due to inter-core communication overheads [8]. To overcome the problem of cache coherence systems, log-based methods are proposed.

The log-based algorithms are a suitable solution for the update-heavy data structure because they allow update operations to proceed with a coarse-grained update lock or without update locks. The benefit of avoiding a fine-grained update lock can eliminate the overhead of acquiring a lock that requires fetching the lock's cache line. Thus, it reduces the cache communication traffic;a contended cache line on many-core processors can take hundreds of cycles to fetch from a remote core [24], and these techniques can be easily applied to other data structures.

One notable recent research accomplishment regarding log-based approach is the OpLog proposed by S. Boyd-Wickizer *et al.*. The OpLog utilizes representative distributed systems management concepts(e.g., Google spanner's synchronized clocks scheme [25]) and proposes the log-based algorithm to shared-memory systems. The OpLog shows significant improvement in performance scalability for update-heavy operating system data structures. Though the OpLog forms an important basis for another step of improvement in performance scalability problem in many core systems, it still has limitation that it's synchronized time-stamp counters might cause additional overhead during log management process.

III. Design

The LDU is a log-based concurrent update method to remove scalability bottlenecks for the update-heavy data structure. The LDU further improves the performance scalability for update-heavy data structures by eliminating the synchronized time-stamp management overhead from previous research. One additional advantage of using the LDU is that it reuses the log slots already allocated and used by preceding operations. This section explains these algorithmic design aspects of the LDU.

A. Approach

The fundamental reason for requiring synchronized time-stamp counters is that some operations need to be ordered. For example, a process logs an insert operation to the per-core memory, then it migrates to another core, and it logs a remove operation, which must eventually execute after the insert operation [8]. To specifically explain a time-sensitive log, we use the symbol label used by this paper [15]. We describe insert as plus-circles ⊕, remove as minus-circles ⊖ and object as color-circles (object B). Color and vertical offset differentiate cpus. For example

$$\oplus_{A},\ \oplus_{B},\ \oplus_{C},\ominus_{A},\ \ominus_{C},\ \oplus_{A},\ \oplus_{C},\ominus_{C}$$

consists of five insert operations, three remove operation, three cpus, and three objects. This example shows that $\oplus_{A}$ and $\ominus_{A}$, time-sensitive logs, must be executed in chronological order. The LDU can eliminate this time-sensitive logs at update time;as a result, the synchronized time-stamp counters are eliminated. One more important fact that these time-sensitive operation logs may be removed by optimization phase. For example, insert-remove operations or remove-insert operations, $\oplus_{A}\ominus_{A}$, $\oplus_{C}\ominus_{C}$ and $\oplus_{C}\ominus_{C}$, are the cancelable operations before a reader, so the remained operation logs,

$$\oplus_{B},\ \oplus_{A}$$

, are non-time-sensitive logs. When update operations occur, the LDU removes these time-sensitive logs using the update-side removing technique.

To remove the time-sensitive log, the LDU uses the update-side removing scheme. When insert-remove operations occur in terms of the same object, the scheme removes the insert-remove operations at update time. The OpLog also optimizes by removing the existing operation rather than adding the new one, but the OpLog can not remove log when a thread migrates other core, so it needs additional sequential processing during the optimization phase. The LDU, however, has no problem with the migrating other core during logging process because it uses the shared atomic swap operation in an individual object.

The update-side removing logs scheme performs an atomic swap operation in an individual object for shared memory systems. This atomic swap operation allows update operations to atomically remove with the previous cancelable log. To achieve update-side removing log, first, add the mark field to the individual object structure then use it as a status flag. For example, consider the insert-remove operation sequence at the same object. The first insert operation marks the mark field and then it logs into the queue. If the remove operation occur, the LDU will not logs;it only changes the mark field. When the reader applies logs, the LDU applies the logs to the original data structure in case of the true value of the mark field. The benefit of this update-side removing scheme is twofold:not only it can eliminate the time-sensitive operation but also it can cancel the previous operation log in the queue.

The second technique, called reusing garbage logs, reuses the garbage log instead of creating a new log. The garbage log is the log that has been already cancelled by using the update-side removing scheme, but it still occupies a slot in the queue. For example, consider the insert-remove-insert operations. After the second remove operation, the log is remaining in the queue, but it has been already cancelled by using update-side removing; hence *insert* mark field is zero in the queue. In this case, the third insert operation reuses the log in the queue instead of creating a new log using the atomic swap, so it can reduce not only the memory overhead but also the queuing overhead.

In addition to the update-side removing logs and the reusing garbage logs, the LDU uses the previous research's scheme that periodically applies the operation logs to reduce memory usage and to keep the log from growing without end. This approach is similar with the method of previous research such as the OpLog's batching updates and flat combining(FC)'s combiner thread.

Furthermore, we design a log's queue using both the per-core and the global queue to further support various data structures. The per-core queue of the LDU can remove CAS operations, which access global head pointer. However, the per-core queue does not properly apply to all data structures since it has some drawbacks. The per-core queue has a memory overhead, and developers may need an additional management code for per-core memory management(see section IV). To overcome these shortcomings, the LDU also supports a global queue. The global queue of the LDU is simple and easy to apply, so it can easily use any data structure. Though global queue can not perfectly remove global CAS operations, it can mitigate the cache communication overhead by using update-side removing logs and by reusing garbage logs because it reduces the number of inserting the queue(see section V-B).

The LDU uses the non-blocking queue regardless of the per-core queue or the global queue since it can proceed without any lock. Among non-blocking queues, the LDU uses multiple producers and single consumer based non-blocking queue thereby reducing the CAS operations. This queue always inserts node where is a first pointer, so it can minimize iteration loops because it causes another CAS operation. In addition, this queue merely considers single consumer when it applies the logs, so it does not need complex algorithms for the remove operation. The queue uses atomic swap operation to acquire all operation logs.

B. Example

Figure 2 shows an example of the LDU with a per-core queue and a global queue. In order to explain the concurrent deferred update method for the update-heavy data structure,

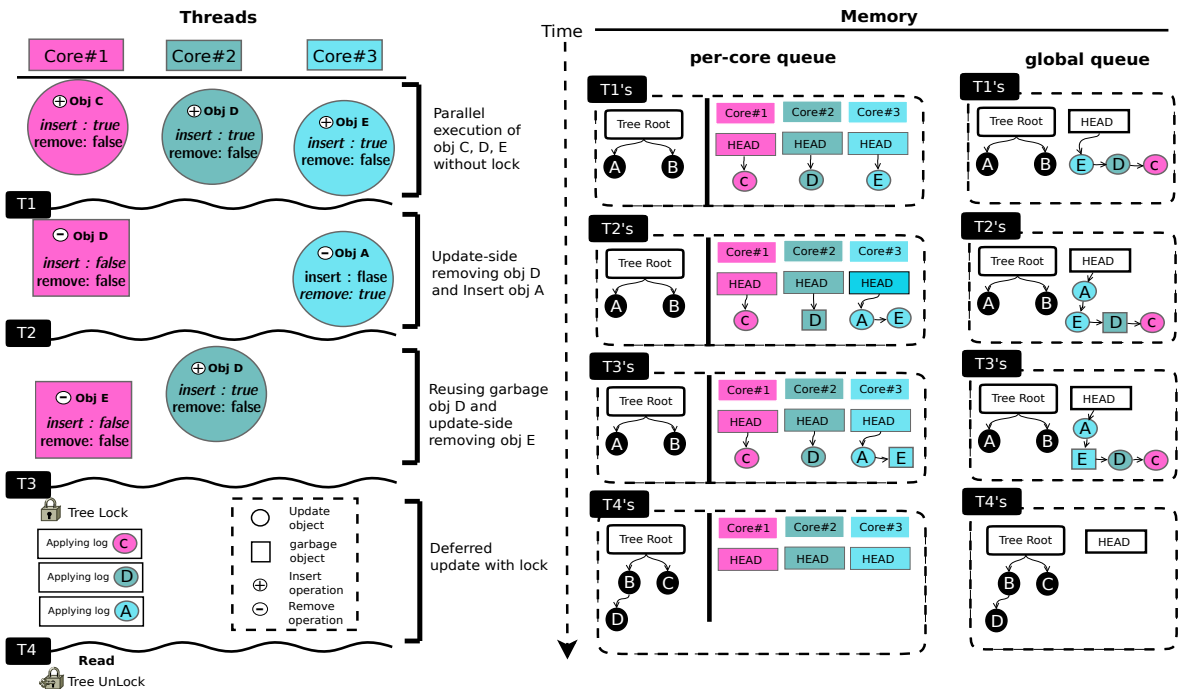


Figure 2: The LDU example showing seven update operations(insert C, insert D , insert E, remove D, insert A, insert D, and remove E) and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2, T3 and T4, respectively. The initial tree data structure contains two object, A and B , and the queue is empty. The seven update operations concurrently execute without locks and a single reader executes with lock.

we show how seven update operations can concurrently execute without lock before the read operation. The update operations sequence is

$$\oplus C,\ \oplus D,\ \oplus E, \ominus D,\ \ominus A,\ \oplus D,\ \ominus E.$$

. To explain the LDU, we use the previously used symbols with a new garbage symbol as rectangle D. In this figure, execution flows from top to bottom. The left side of figure shows cpu operations, and the right side of figure shows data structures contents at a particular time in a memory. Initially, the tree data structure contains A and B, and the queue is empty.

In the top of figure, `Core1`, `Core2` and `Core3` perform the concurrent update operations, $\oplus C$, $\oplus D$ and $\oplus E$, without lock. Since the LDU uses non-blocking queue to save the operation logs, this step does not need a update lock, so all threads can be executed in parallel without any lock contention. At `T1`, the tree contains objects A and B. The per-core queue and the global queue contain $\oplus C$, $\oplus D$ and $\oplus E$, but the logs in the per-core queue are separated.

The next operations are $\ominus D$ and $\ominus A$. When the $\ominus D$ operation is executed, the LDU atomically changes the mark field in the object instead of inserting the queue. Then, $\ominus A$ inserts the queue because it is a new operation log. At `T2`, the per-core queue and the global queue contain $\oplus C$, $\oplus D$, $\oplus E$ and $\ominus A$. The insert mark field in the object D is false, called a garbage object, has remained in the queue, but it was already canceled by update-side removing logs.

The last operations are $\oplus D$, $\ominus E$. The LDU reuses the log in the queue instead of creating a new log using atomic swap. Thus, the object D changes D, and then the object E changes E by performing the update-side removing logs scheme. At `T3`, per-core queue contains $\oplus C$, $\oplus D$, $\ominus A$ and $\oplus E$.

Before the read function, it need to lock the original tree lock using the exclusive lock in order to protect the tree operations. The LDU migrates from queue to tree, each of which is the marked object. Thus, $\oplus C$, $\oplus D$ and $\ominus A$ are migrated except for the $\oplus E$ a garbage log. At T5, the tree contains B, C and D, so finally, the reader can read eventually consistent data.

C. The Algorithm and Correctness

This section shows skeleton of an algorithm. We exclude the log's queue and the LDU's detailed data structures for exposition simplicity.

1) inserting logs: Figure 3 shows concurrent update functions. The concurrent update functions are divided into three phase. The first phase checks this object to see whether or not the object is a cancelable object(Line 4, 20). When this code is executed, the `synchronize` function can be invoked by a reader or a periodic timer, so phase 1 needs the atomic operation. If the corresponding mark field is true,

```
1  bool ldu_logical_insert(struct object_struct *obj,
2      void *head) {
3    // Phase 1 : update-side removing logs
4    if (SWAP(&obj->ldu.remove.mark, false) == false){
5      ASSERT(obj->ldu.insert.mark);
6      obj->ldu.insert.mark = true;
7      // Phase 2 : reusing garbage log
8      if (!TEST_AND_SET_BIT(LDU_INSERT,
9          &obj->ldu.used)){
10       // Phase 3(slow-path): insert log to queue
11       // ... : save argument and operation
12       ldu_insert_queue(head, log);
13     }
14   }
15 }
16
17 bool ldu_logical_remove(struct object_struct *obj,
18     void *head) {
19   // Phase 1 : update-side removing logs
20   if (SWAP(&obj->ldu.insert.mark, false) == false){
21     ASSERT(obj->ldu.remove.mark);
22     obj->ldu.remove.mark = true;
23     // Phase 2 : reusing garbage log
24     if (!TEST_AND_SET_BIT(LDU_REMOVE,
25         &obj->ldu.used)){
26       // Phase 3(slow-path): insert log to queue
27       //... : save argument and operation
28       ldu_insert_queue(head, log);
29     }
30   }
31 }
```

Figure 3: The LDU concurrent update algorithm. This logging may be called by original update functions without locks. The concurrent update functions are divided into three phase.

then its mark field is changed to false. In phase 2 checks this log to see weather or not has already inserted in the queue(Line 8, 24). If so, because mark field is marked(Line 6, 22), this function directly returns true. In the last phase, the operation log inserts the non-blocking queue when the operation log is the first used log(Line 12, 28).

This algorithm is correct because Linux kernel has a unique update operations sequence. For example, if an insert operation occur, then next operation must be a remove operation at the same object because the kernel's update function is separated from search, alloc and free functions. The remove-remove or insert-insert operation in Linux kernel is forbidden: if remove-remove operation occur, the second remove operation may encounter a crash because this object can be concurrently freed after the first remove operation, so we check the corresponding mark field(Line 5, 21).

2) applying logs: Figure 4 shows deferred update function, which applies the operation logs. The `synchronize` function is invoked before the read, or it can be periodically invoked by the timer handler because of preventing the continuous growing the logs. Before the execution of the `synchronize` function, it has been locked by using the object lock, so this function proceed with a single consumer thread in a way similar to the OpLog's batching updates and FC's combiner thread. First, the `synchronize` function

```
1  void synchronize_ldu(void *head)
2  {
3    entry = SWAP(&head->first, NULL);
4    //iteration all logs
5    for_each_all_logs(log, entry, next) {
6      //... : get log's arguments
7      //atomic swap due to update-side removing
8      if (SWAP(&log->mark, false) == true)
9        ldu_apply_log(log->op_num, log->args);
10     CLEAR_BIT(log->op_num, &obj->ldu.used);
11     // once again check due to reusing garbage logs
12     if (SWAP(&log->mark, false) == true)
13       ldu_apply_log(log->op_num, log->args);
14   }
15 }
```

Figure 4: The LDU applying logs algorithm. The `synchronize_ldu` may be called by a reader or a timer handler and converts logs to original data structure traversing the log's queue.

acquires queue's head pointer by using atomic swap operation(Line 3). Because the LDU periodically applies the logs queue, the LDU update operations may concurrently execute with the `synchronize` function. Thus, before the applying original data structure, the mark field is set to false(Line 8, 9). The used flag for the garbage log is set to false indicating that the queue does not contain this object(Line 10). The `synchronize` function once again checks(Line 12, 13) to see whether the mark field is changed between the applying logs(Line 8) and clearing garbage bit(Line 10).

IV. Applying Linux kernel

This section shows how to apply the LDU to the complex Linux virtual memory system to solve the update serialization problem;it deals with more practical one.

The Linux reverse page mapping(rmap), a kernel memory management mechanism, consists of anonymous rmap and file rmap, are a update-heavy data structure. These two rmaps maintain virtual address(VMAs) to translate physical addresses to virtual address [26], and the rmaps are a shared global resource between processes. These global resource of rmap are managed by using a interval tree. To protect these shared tree, Linux kernel uses the reader-writer semaphore, and simultaneous creation of many processes becomes bottlenecks because not only the rmap's update operations can not run in parallel but also update's lock brings about the cache invalidation traffic. On the contrary, the rmap rarely reads the interval tree when it swaps a physical page out to disk, migrates other cpu, or truncates a file.

A. Anonymous mapping

Figure 5 shows the anonymous rmap data structure. When a process spawns, the parent's anonymous vma chain(AVC) are copied to a child, and then a new anonymous vma(`struct anon_vma`) is created. When a process simultaneously spawns, the more complex anonymous rmap data structures are created;the anonymous ramp is one of the

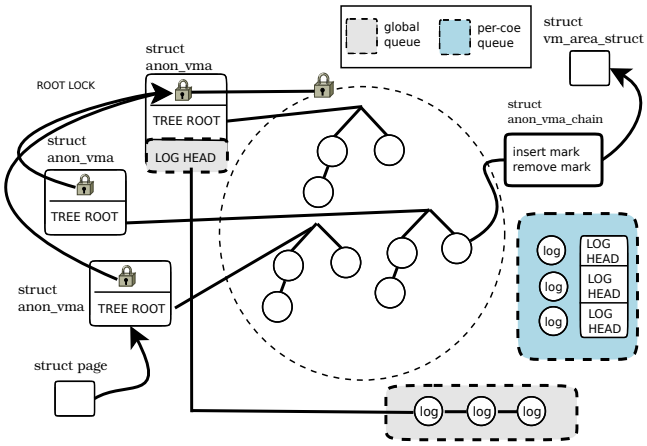


Figure 5: An example of applying the LDU to anonymous reverse mapping. When a process simultaneously spawns, the root lock leads to a scalability bottleneck. The log's queue is added into the per-core memory or the global head object(`struct anon_vma`)

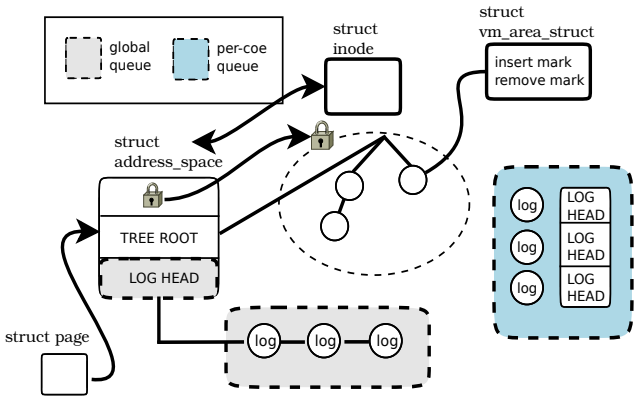


Figure 6: An example of applying the LDU to file reverse mapping. The file rmap can be serialized at updating VMAs into the interval tree. The log's queue is added into the per-core memory or the global head object(`struct address_space`)

complex data structure in Linux kernel [27]. The anonymous rmap uses the root lock since the AVCs are shared with child processes, so this root lock causes a lock contention problem [20].

To eliminate this lock contention problem, we add the insert and remove mark field in the individual object(`struct anon_vma`), and then we implement the update-side removing logs scheme. Understanding the log's position of queue header is important. As noted earlier, since the anonymous rmap uses the root lock, the per-core queue version of the LDU logs into a per-core memory with root information, or the global queue logs into a root data structure(`struct anon_vma`). Consequently, the LDU does not largely modify the original data structure, which shows why the LDU is a lightweight method.

B. File mapping

Figure 6 shows the rmap for file. In order to translate physical addresses to virtual address, the page(`struct page`) indicates the address space object(`struct address_space`), and the address space object manages the VMAs by using the interval tree. This interval tree is a shared resource between processes. Because the system calls such as `fork()`, `exit()` and `mmap()` entail concurrent updating VMAs into the shared resource. when the processes simultaneously invoke these system calls, the file rmap can be serialized at the update operations.

The LDU can easily be applied to the file ramp data structure. For instance, to use the LDU, a developer adds log's queue header into the per-core memory or into the original data structure(`struct address_space`), and then adds mark field to the individual object(`struct vm_area_struct`). Then, the developer modifies update function to logging function without a lock. Finally, the developer creates `synchronize` function and calls the `synchronize` function before the read.

This figure clearly shows why the LDU additionally supports the global queue because it is a simpler and easier scheme because the log's head pointer is located in the interval tree's data structure. On the other hand, the per-core queue may need an additional per-core queue management scheme due to the its isolated memory location.

C. Detail Implementation

Because the log's head pointer for the per-core queue is separated with the original data structure, the implementation of per-core queue uses a per-core hash table method that can allow each object to distinguish. The per-core hash table implemented as a direct-mapped cache, which one bucket only has an object because recently used objects will be in the hash table in a way similar to the OpLog's per-core hash table. When this hash table is met a hash conflict, the LDU evicts the object in the hash slot. Moreover, this method reduces additional tasks of programmers because it can minimize code modifications and does not need an additional lock. The per-core hash table, however, incurs a hash conflict overhead. This method is useful when a number of the root objects are infrequently created like the file rmap(`struct address_space`). On the other hand, since the anonymous rmap severely creates many root objects(`struct anon_vma`), it causes a hash conflict overhead. Therefore, in the case of the anonymous ramp, we did not distinguish object headers, but it needs additional tasks with global lock.

We have implemented the new deferred update algorithm in Linux 4.5-rc6 kernel, and our modified Linux is available as open source. The implementation is stable enough and has passed the testing related with virtual memory, scheduler,

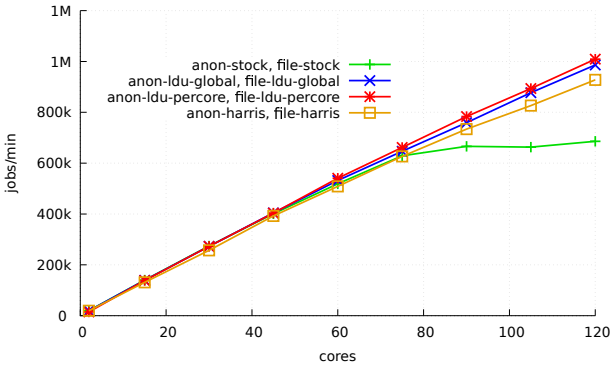


Figure 7: Scalability of AIM7-multiuser. The LDU and the Harris list scale well;in contrast, up to 75 core, the stock Linux scale linearly, then it flattens out.

and file in the Linux Test Project [28].

V. Evaluation

A. Experimental setup

For the purpose of performance evaluation of the proposed LDU technique, we performed experiments using a Linux kernel where LDU technique is implemented compared to a lock-free list version of Linux proposed by Harris [21]. The reason why we used Harris algorithm for our comparison purposes is that the algorithm is considered as the representative concurrent non-blocking algorithm. The basic algorithms of Harris linked list are from sysnchrobench [29] and ASCYLIB [30], and we slightly converted the Harris linked list to be adopted in Linux kernels.

The hardware specification we used for our experiments are a 120 core machine with 8-socket, Intel E7-8870 chips(15 cores per socket) equipped with 792 GB DDR3 DRAM.

We selected benchmark programs with fork-intensive applications since the fork-intensive update-heavy data structure accesses could maximally benefit from the proposed technique. The benchmark programs are AIM7, a Linux scalability benchmark, Exim, an email server in MOSBENCH, and Lmbench, a micro benchmark. The workloads exhibit the high lock contentions because of the two reverse mappings. Moreover, the AIM7 benchmark is widely used in the Linux community not only for testing Linux kernel but also for improving the scalability. The Exim is a real world application, but it has scalability bottlenecks caused by the Linux fork. Finally, in order to only focus on the fork performance and scalability, we selected the Lmbench.

We used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used the LDU version of Linux kernel that used global queue. Next, we used the per-core queue version of the LDU. Finally, we used Harris lock-free list version of Linux kernel as we mentioned earlier. Unfortunately, direct comparison experiments between the LDU and the OpLog was not possible for a few implementation-related issues (e.g., we could not obtain the detailed implementation of the OpLog).

B. AIM7

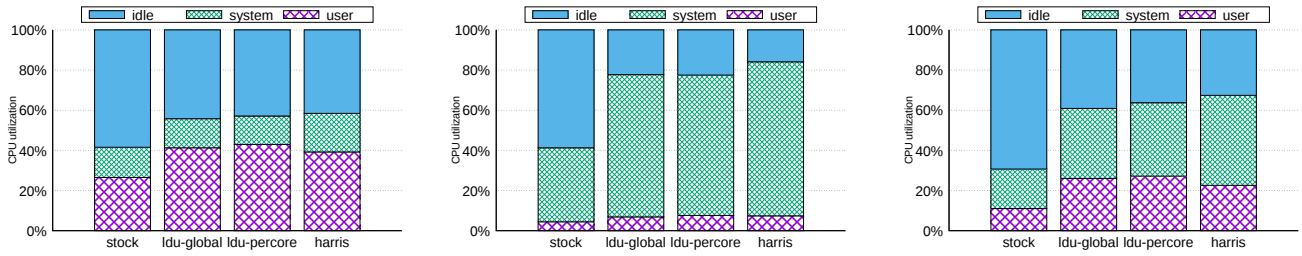
We used AIM7-multiuser, which is one of fork-intensive workload in AIM7. The multiuser workload simultaneously creates many processes with various operations(see section II), and we used the temp filesystem to minimize the file system bottleneck. We increased a number of users in proportion to number of cores.

The results for AIM7-multiuser are shown in Figure 7. Up to 75 core, the stock Linux scales linearly and then serialized updates become bottlenecks. However, up to 120 core, the Harris and our LDU scale well because these workloads can concurrently execute update operations without the reader-writer semaphores(`anon_vma->rwsem`, `mapping->i_mmap_rwsem`). The per-core queue version of the LDU shows the best performance and scalability outperforming stock Linux by 1.5x and Harris by 1.1x. In addition, although the global queue version of the LDU has the global CAS operation, it also has high performance and scalability because the global CAS operations are mitigated by two LDU techniques;it had 2% performance degradation compared with per-core queue version of the LDU. Furthermore, the stock Linux shows the highest idle time(58%)(see figure 8(a)) since it waits to acquire semaphore(i.e., `anon_vma->rwsem`, `mapping->i_mmap_rwsem`). Surprisingly, although two LDU have higher idle time than the Harris version, the throughput shows higher than the Harris because of our efficient algorithm.

C. Exim

To measure the performance of the Exim, we used the MOSBENCH, a many-core scalability benchmark. The Exim email server is designed to scale because the Exim delivers messages to mail boxes in parallel using the Linux process;the Exim is a fork-intensive workload. Clients ran on the same machine and each client sent to a different user to prevent contention on user mail file. The Exim was bottlenecked by the filesystem [1] since the message body appends to the per-user mail file, so we used the separated tmpfs to reduce filesystem bottlenecks.

Results shown in figure 9 indicate that the Exim scales well up to 60 core, but the stock Linux performance decreases near 60 core. Both the Harris and the LDU increase up to 105 core because they can execute concurrent updates without the semaphores. The per-core queue version of the LDU performs better due to the fact that it can reduce cache coherence-related overheads outperforming stock Linux by 2.6x and Harris by 1.2x. Even though we applied the scalable solutions, the Exim shows the limitation near 105 core since the Exim process has a relatively large size of virtual address mapping that leads to the clearing virtual address



(a) AIM7 - 120core (b) Exim - 120core (c) Lmbench - 120core

Figure 8: CPU utilization on 120 core. The stock Linux has the highest idle time since it waits to acquire semaphore(`anon_vma->rwsem`, `mapping->i_mmap_rwsem`). The LDU has higher idle time than the Harris linked list because of LDU's efficient algorithm.

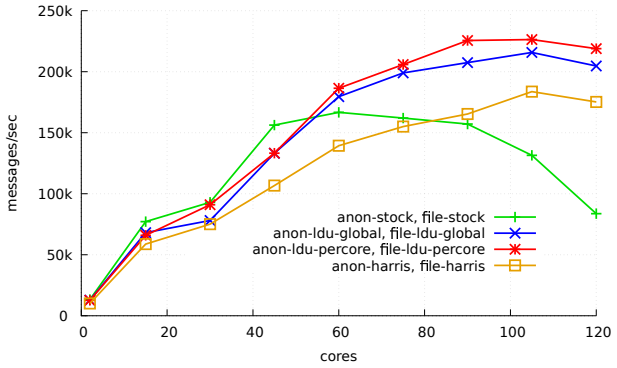


Figure 9: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both the Harris list and our the LDU go up to grow up to 105 core.

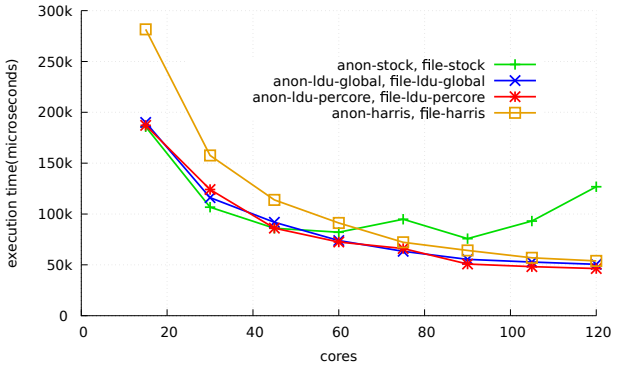


Figure 10: Execution time of the process management workload in the Lmbench. The benchmark drops down for all methods up to 60 core but either goes up or goes down slightly.

mappings overheads and many soft page fault during the process destruction which executes more slowly with more remote socket memory access. The Harris has 15% idle time, whereas per-core queue version of the LDU has 22% idle time because the LDU has the efficient algorithm(see figure 8(b)).

D. Lmbench

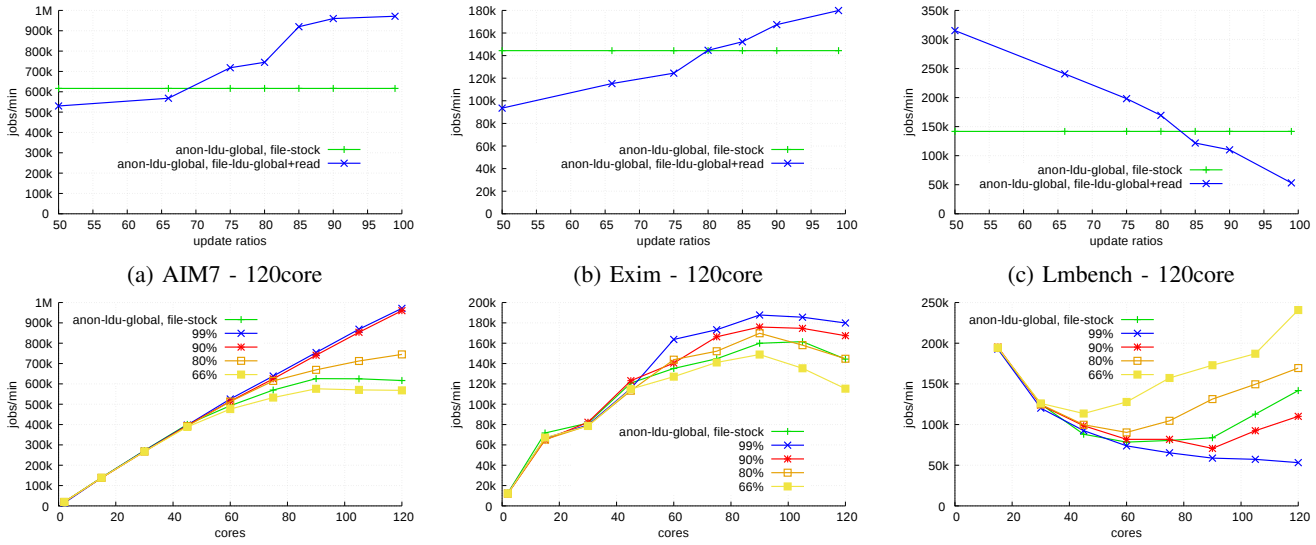
The Lmbench has various micro benchmarks including process management workloads. We used the process management workload in the Lmbench. This workload is used to measure the basic process primitives such as creating a new process, running a program, and context switching. We configured process create workload to enable the parallelism option(the value was 1000).

The results for the Lmbench are shown in Figure 10, and the results show the execution times. Up to 45 core, the stock Linux scales linearly and then the execution time goes up to grow. The per-core version of the LDU outperforms the stock Linux by 2.7x and the Harris by 1.1x at 120 core. While the stock Linux has 69% idle time, other methods have approximately 35% idle time since the stock Linux waits to acquire two rmap semaphores(`anon_vma->rwsem`, `mapping->i_mmap_rwsem`)(see figure 8(c)). Indeed, our main motivation in the LDU is to improve the performance and scalability on the many-core systems, so we did not consider a low cores performance(under 30 cores). However, up to 30 core, our LDU is similar to the stock Linux performance, but the Harris has low performance up to 60 core.

E. Updates ratio

One question that could be raised regarding the proposed LDU scheme would be how the performance scalability is affected by the frequency of read operations since the proposed technique has only focused on update-heavy data structures with very low read ratios. Even read operations could be slower since read operation should perform codesynchronize function to apply logs.

To understand the effect of read operations, we performed another experiment with intentionally adding the read operations in proportion to update operations. The anonymous rmap used the global queue version of the LDU, and then we sequentially increased read (`lock`, `synchronize`) ratios regarding the file rmap. The upper graphs of figure 11 shows the performance on 120 core depending on its update ratios, and the lower graphs represent the scalability.



(a) AIM7 - 120core (b) Exim - 120core (c) Lmbench - 120core

(d) AIM7 - scalability (e) Exim - scalability (f) Lmbench - scalability

Figure 11: Performance depending on update ratios and scalability. The scalability of AIM7 shows that the LDU has substantially high scalability at the 90% and the 99% update rates as well as the 80% update rates. The Exim and the Lmbench show that the LDU outperforms the stock kernel after 85%.

Since the AIM7 has less fork-intensive workload than other ones, the read operations are invoked relatively infrequently. As a result, although the data structure uses 75% update rates(3 update, 1 read), the LDU version of Linux has outstanding performance than stock Linux. The scalability of AIM7 shows that the LDU has substantially high scalability at the the 90% and the 99% update rates as well as the 80% update rates.

The Exim and the Lmbench show the extremely high fork-intensive workload. As a result, the stock Linux outperforms the LDU by approximately 80%, but the LDU outperforms the stock kernel after 85%. This explains the LDU has outstanding performance even when the read operations frequently occur.

VI. Related work

Operating systems scalability. To improve the scalability, researchers have attempted to create new operating systems [16] [31] or have attempted to optimize existing operating systems [1] [24] [32] [8] Our research belongs to optimizing existing operating systems in order to solve the Linux fork scalability problem. However, previous research did not deal with the anonymous reverse mapping, which is one of the fork scalability bottleneck.

Scalable lock. Scalable locks have been designed by the queue-based locks [33] [34], hierarchical locks [35] [36] and delegation techniques [7] [37] [38]. Our research is similar to the delegation techniques because the LDU's `synchronize` function runs as a combiner thread;it improves cache locality. However, our approach not only can improve cache locality but also can eliminate synchronization methods during updates due to using a lock-free manner.

Scalable data structures. Many scalable data structures with scalable schemes show different performances depending on their update ratios. In low and middle update rate, researchers have attempted to create new scalable schemes [39] [4] [21] or have attempted to adapt these scheme to data structures [5] [6] [24]. In high update rate, the OpLog shows significant improvement in performance scalability for update-heavy data structures in many core systems, but suffers from limitation and overhead due to time-stamp counter management. We substantially extend our preliminary work [40] not only to support per-core algorithm but also to apply the LDU to anonymous rmap due to improving the Linux kernel scalability.

VII. Conclusion and Discussion

We proposed and evaluated a novel concurrent update method, LDU, to maximally improve the performance scalability of Linux kernel in many core systems. Such improvement was possible through eliminating the synchronized time-stamp counters management overhead found in a previous well-known scheme, OpLog. Our experiments using a Linux kernel with our LDU implementation revealed that the proposed LDU shows better performance up to 2.7 times stock Linux kernel on the 120 core machine.

While the proposed technique achieves significant improvement in performance scalability through eliminating time-sensitive logs, there still remain the data structures to consider to further improve the scalability (i.e., stack and queue data structures). Future direction of research is to create a new synchronization scheme by combining two techniques(the LDU and the OpLog) to support the stack and queue.

The LDU is implemented on to Linux kernel 4.5-rc6 and available as open-source from https://github.com/manycore-ldu/ldu.

REFERENCES

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010, pp. 1–16.

[2] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding Manycore Scalability of File Systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Jun. 2016, pp. 71–85.

[3] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" 2011.

[4] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 168–183.

[5] M. Arbel and H. Attiya, "Concurrent Updates with RCU: Search Tree As an Example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14, 2014, pp. 196–205.

[6] M. Dodds, A. Haas, and C. M. Kirsch, "A Scalable, Correct Time-Stamped Stack," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, 2015, pp. 233–246.

[7] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat Combining and the Synchronization-parallelism Tradeoff," ser. SPAA '10, 2010, pp. 355–364.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "OpLog: a library for scaling update-heavy data structures," in *Technical Report MIT-CSAIL-TR2014-019*, 2014.

[9] P. McKenney, "Some more details on Read-Log-Update," 2016, https://lwn.net/Articles/667720/.

[10] A. Morrison, "Scaling synchronization in multicore programs," *Queue*, vol. 14, no. 4, pp. 20:56–20:79, Aug. 2016.

[11] "AIM Benchmarks," http://sourceforge.net/projects/aimbench.

[12] "Exim Internet Mailer," 2015, http://www.exim.org/.

[13] "MOSBENCH," 2010, https://pdos.csail.mit.edu/mosbench/mosbench.git.

[14] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable Tools for Performance Analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[15] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 1–17.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 43–57.

[17] D. Bueso, "Scalability Techniques for Practical Synchronization Primitives," vol. 58, no. 1, Dec. 2014, pp. 66–74.

[18] D. Bueso and S. Norto, "An Overview of Kernel Lock Improvements," 2014, http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[19] "LOCK STATISTICS," https://www.kernel.org/doc/Documentation/locking/lockstat.txt.

[20] T. C. Andi Kleen, "Scaling problems in Fork," in *Linux Plumbers Conference, September*, 2011.

[21] T. L. Harris, "A Pragmatic Implementation of Non-blocking Linked-Lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01, 2001, pp. 300–314.

[22] M. Fomitchev and E. Ruppert, "Lock-free Linked Lists and Skip Lists," in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '04, 2004, pp. 50–59.

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free Linked-lists," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 309–310.

[24] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable Address Spaces Using RCU Balanced Trees," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012, pp. 199–210.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[26] D. McCracken, "Object-based reverse mapping," in *In Proceedings of the 2004 Ottawa Linux Symposium*, 2004.

[27] J. Corbet, "The case of the overly anonymous anon_vma," 2010, https://lwn.net/Articles/383162/.

[28] "Linux test project," https://github.com/linux-test-project/ltp.

[29] V. Gramoli, "More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, 2015, pp. 1–10.

[30] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015, pp. 631–644.

[31] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "An Operating System for Multicore and Clouds: Mechanisms and Implementation," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010, pp. 3–14.

[32] A. T. Clements, M. F. Kaashoek, and N. a. Zeldovich, "RadixVM: Scalable Address Spaces for Multithreaded Applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 211–224.

[33] 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.

[34] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue Locks on Cache Coherent Multiprocessors," in *Proceedings of the 8th International Symposium on Parallel Processing*. IEEE Computer Society, 1994, pp. 165–171.

[35] Z. Radovic and E. Hagersten, "Hierarchical Backoff Locks for Nonuniform Communication Architectures," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. IEEE Computer Society, 2003, pp. 241–.

[36] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, Locality-preserving Locks," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, 2016, pp. 22:1–22:14.

[37] P. Fatourou and N. D. Kallimanis, "Revisiting the Combining Synchronization Technique," *SIGPLAN Not.*, vol. 47, no. 8, pp. 257–266, Feb. 2012.

[38] D. Klaftenegger, K. Sagonas, and K. Winblad, "Delegation Locking Libraries for Improved Performance of Multithreaded Program," in *Euro-Par 2014 Parallel Processing*, 2014, pp. 572–583.

[39] P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," in *Parallel and Distributed Computing and Systems*, October 1998, pp. 509–518.

[40] J. Kyong and S.-S. Lim, "LDU: A Lightweight Concurrent Update Method with Deferred Processing for Linux Kernel Scalability," in *Proceedings of the IASTED International Conference, Parallel and Distributed Computing and Networks (PDCN 2016)*, 2016.