A Lightweight Log-based Deferred Update for Linux Kernel Scalability

Abstract—**We propose a novel light weight concurrent updates method, LDU, to improve performance scalability for Linux kernel on many-core systems through eliminating lock contentions for update-heavy global data structures during process spawning and optimizing update logs. The proposed LDU is implemented into Linux kernel 4.5 and evaluated using representative benchmark programs. Our evaluation reveals that the Linux kernel with LDU shows performance improvement by ranging from Xx through Xx on a 120 core system.**

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

I. Introduction

With increasing core counts, processors move from multi-core to many-core systems. In many-core systems, parallelism of operating system kernel has been an important part of system parallelism. If the kernel doesn't scale, applications depending on the shared resources provided by the operating system kernel won't scale[]. Linux kernel has been naturally considered to use a many-core operating system. The Linux kernel, however, has been scalability problems [1] [2]. One of scalability problem is update serizlization due to the update lock contention;update operations cannot run in parallel [5] [3] [4].

In order to solving the update serialization problem, various concurrent updates methods are proposed [6] [3]. These methods show different performance at update ratio. In case of a high update rates(update-heavy) data structure, the update serialization problem can pose a scalability bottleneck. Log-based algorithms [8] [9] can solve this update serialization problem to reduce cache coherence-related overheads for update-heavy data structure. Log-based algorithm is that when update operations occur, it logs the update operation and applies the all operation logs to the data structure before read operation, so reader can read up to date data structure;it similar to CoW(Copy On Write) [10].

S. Boyd-Wickizer et al. proposed Oplog [9] where logs update operations with synchronized timestamp counters to solve concurrent updates and cache communication bottleneck for update-heavy data structure. Synchronized timestamp counters based method obtains outstanding update-side scalability because they can record their operation logs without cache communication overhead because it logs per-core memory. However, synchronized timestamp counters method may incur timestamp merging and ordering process . When increasing core counts, resolving logs(merging, absorbing) may require additional sequential processing, which can limit scalalbility and performance.

To solve the sequential processing due to the synchronized timestamp counters for shared memory system, we propose a novel lightweight log-based deferred update method. LDU simply removes operation log requiring timestamp counter at update time and reuses the garbage log in log's queue without creating new log. Therefore, we can eliminate synchronized timestamp counter and cache communication bottleneck. We combine both log-based concurrent updates that is widely used in distributed system and minimal hardware-based synchronization method(compare and swap, test and set, atomic swap) that is used in shared-memory system.

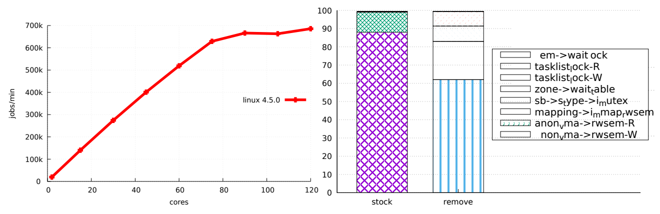
LDU contains not only benefits of log-based algorithms but also additional benefit. First, log-based algorithms can remove synchronization overhead resulted in reducing the cache invalidation traffic. Second, these techniques can be easily applied to other data structures. In addition, finally, since LDU can eliminate log before inserting log's queue, LDU's log management is efficient.

To evaluate our approach, we adopted LDU in two Linux kernel reverse page mapping(anonymous page mapping, file page mapping) resulting in fork scalability bottleneck in Linux kernel because of high update rates global data structure. In addition, we implemented the LDU in a Linux 4.5.rc4 with elimination of synchronization method. We evaluated the performance and scalability using a fork-intensive workload- AIM7 [11], Exim [12] from MOSBENCH [13], lmbench [14]-our design improves throughput and execution time on 120 core by 1.7x, 1.6x, 2.2x respectively, relative to stock Linux.

Contributions. This paper makes the following contributions:

- We develop a novel lightweight log-based deferred update method. LDU simply removes operation log requiring timestamp counter and reuses the garbage log in log's queue;therefore, LDU can solve the Oplog's sequential processing problem due to the synchronized timestamp counters.
- We adopted LDU in Linux kernel two reverse mapping(anonymous, file) on practial 120core system to reduce fork scalability bottleneck. Our design improves throughput and execution time on 120 core by 1.7x 2.2x.

The rest of this paper is organized as follows. Section 2 describes the background and Linux scalability problem. Section 3 describes the design of the LDU algorithm and Section 4 explains how to apply to Linux kernel. section 5



(a) aa (b) aa

Figure 1: Scalability of AIM7 multiuser. This workload simultaneously create many processes. Up to 60 core, the stock Linux scale linearly, then they flattens out.

explains our implementations in Linux and Section 6 shows the results of the experimental evaluation. Finally, section 8 concludes the paper.

II. Background and Problem

Operating system paralleism is of utmost importance for scalable systems Aplications performance would be limited by the operating system when the operating system doesn't scale [15][Corey]. Because Linux has heavily optimized for multi-core operating system, we examine AIM7-multiuser benchmark on Linux. AIM benchmark has, until recently, been heavily used in research area and Linux community [16] [17]. AIM7-multiuser workload simultaneously create many processes with disk-file operations, virtual memory operations, pipe I/O, and arithmetic operation. To minimaize file system bottleneck, we used temp filesystem [18]. The Figure-xx(a) show the scalability when running AIM7. Up to 60 core, the stock Linux scale linearly, then they flattens out.

To understand the source of scalability bottleneck on 120core, we profile a lock contention using lock_stat, a Linux kernel lock profiler that reports how long each lock is held and the wait time to acquire the lock. The Figure-xx (a) show the cost of acquiring the lock when running AIM7 on a stock Linux kernel on the Intel 120 core machine. Using AIM7 benchmark, the stock Linux extremely high contend on anonymous reverse page's read-writer sempaphore(anon_vma-¿rwsem). This lock contention generated by creating the number of process simontanously with Linux fork. These reverse page mapping records page information when using fork(), exit() and mmap() system call to find all page table entries. To understand the other bottleneck, we conducted a workaround by removing anonymous rmap relative code in the part of the fork code with page-swap off avoding the Linux page reclaiming. When removing the anon_vma, Linux was contended on file reverse page mapping(i_mmaping-rwsem).

Resulting from our research, both anonymous reverse page mapping reporting from Linux community[] and file reverse page mapping reporting from S. Boyd-Wickizer[] are a significant factor in a fork scalability problem. Thus, in order to perfect scalability of the fork, both the file reverse mapping and the anonymous reverse mapping should execute concurrently without lock. More specifically, the fundamental scalability problem of reverse mapping is their serialized updates operation because operating system kernel are serialized at the updates operation.

In order to achieve scalable concurrent updates allowing update operations to proceed without update locks, both the non-blocking algorithms [28] [35] [36] and log-based algorithms are proposed. In non-blocking algorithms, when they observed against the current value in global data structure such as list, queue and tree, they execute a CAS to compare the against value. When the value has been overridden, they must be retried later on. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails will result in bottlenecks due to inter-core communication overheads [9]. To overcome the issues caused by cache coherence systems, log-based methods are proposed. This paper presents a log-based deferred design that alows concurrent updates to scale, so that multiprocessed applications can scale to large numbers of cores.

III. LDU Design

The LDU is a log-based concurrent updates method to remove scalability bottlenecks for update-heavy data structure. Synchronized timestamp counters method may incur timestamp merging and ordering process . When increasing core counts, resolving logs(merging, absorbing) may require additional sequential processing, which can limit scalalbility and performance. To solve the sequential processing due to the synchronized timestamp counters for shared memory system, we propose a novel lightweight log-based deferred update method. LDU simply removes operation log requiring timestamp counter at update time and reuses the garbage log in log's queue without creating new log. Therefore, we can eliminate synchronized timestamp counter and cache communication bottleneck. This section explains these algorithmic design aspects of LDU.

A. *Log-based Concurrent updates*

Log-based algorithms [8] [9] can solve this update serialization problem to reduce cache coherence-related overheads for update-heavy data structure. Log-based algorithm is that when update operations occur, it logs the update operation and applies the all operation logs to the data structure before read operation, so reader can read up to date data structure;it similar to CoW(Copy On Write) [10].

Log-based approaches can help to support high update rates because

LDU contains not only benefits of log-based algorithms but also additional benefit. In addition, finally, since LDU can eliminate log before inserting log's queue, LDU's log management is efficient. Second, these techniques can be easily applied to other data structures. First, log-based algorithms can remove synchronization overhead resulted in reducing the cache invalidation traffic.

B. Approach

The fundamental reason for requiring synchronized timestamp counters is that some operations should need to be ordered. For example, a process logs an insert operation to per-core memory, then migrates to another core, and logs a remove operation to per-core memory. The remove operation must eventually execute after the insert operation[]. Thus, OpLog should needs synchronized timestamp counters. To explain this time-sensitive log, we use 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

⊕Ⓐ, ⊕Ⓑ, ⊕Ⓒ,⊖Ⓐ, ⊖Ⓒ, ⊕Ⓐ, ⊕Ⓒ,⊖Ⓒ

consists of five insert operations, three remove operation, three cpus, and three objects. This example show that ⊕Ⓐ and ⊖Ⓐ, time-sensitive log, must be executed in chronological order. LDU can eliminates this time-sensitive log when logs update operations with time stamps, so LDU can removes synchronized timestamp counter. 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 such as ⊖Ⓑ, ⊕Ⓒ are cancelable operations before the read. Eventually remaining operation logs such as

⊕Ⓑ, ⊕Ⓐ

, which are non-time-sensitive logs. When updates operation occurs, LDU removes these time-sensitive log using update-side removing technique.

To remove the time-senstivive log, LDU uses the update-side removing which is if insert-remove operation occur in terms of same object, the same object's insert-remove operation would be removed on the update time. Indeed, the OpLog also optimizes by removing the existing operation rather than adding the new one. OpLog optimization is that becasue it may migrates other core, and then it logs per-core memory, it needs to merge and search for removing the operation log, which is additional sequntial processing for optimization. LDU, however, have no problem migrating other core

To the best of our knowledge, LDU is the first log-based concurrent updates to remove logs on the update time because we found that Linux kernel's updates operations involved sequence of operations. The Linux kernel, for example, if insert operation occur, then next operation must be remove operation becasue the kernel's update function has separated from search, alloc and free functions. Therefore, 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 freed by first remove operation concurrently. Therefore, the design of the LDU is inspired by these kernel update operations sequeunce.

Update-side removing logs are implemented by performing atomic swap operation in an individual object for shared memory systems. This atomic swap operation alows update operations to atomically remove with the previous cancelable log. To that these LDU performs adding the marking field in the object structure. For example, consider the insert-remove operation sequence at same object. The first insert operation marks the feld of mark and then the log inserts queue. After the next remove operation, LDU does not logs and proceed with changing the mark feld. The reader applies logs, marking feld is true, to the original data structure. The benefit of this update-side removing is twofold:not only it can eliminates the time-sensitive operation but also it can cancels the previous operation log in the queue with current operation.

The second optimization, called reusing garbage logs, reuse garbage log, whose log remained queue and aleady canceled through the update-side removing.

For example,

To remove dependency of queue, LDU logs in non-blocking queue.

LDU design using both global queue and per-core queue.

To reduce memory usage and to keep the log from growing without end, LDU periodically apply the time-ordered operations at the head of the log to remove these opearation from the log.

C. LDU example

D. The LDU Algorithm and Correctness

1) logical update: inserting logs:

2) physical update: applying logs:

IV. Concurrent updates for Linux kernel

A. Case study:reverse mapping

B. file mapping

C. anonymous mapping

V. Implementation

VI. Evaluation

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?
- What about LDU's read-write ratio?

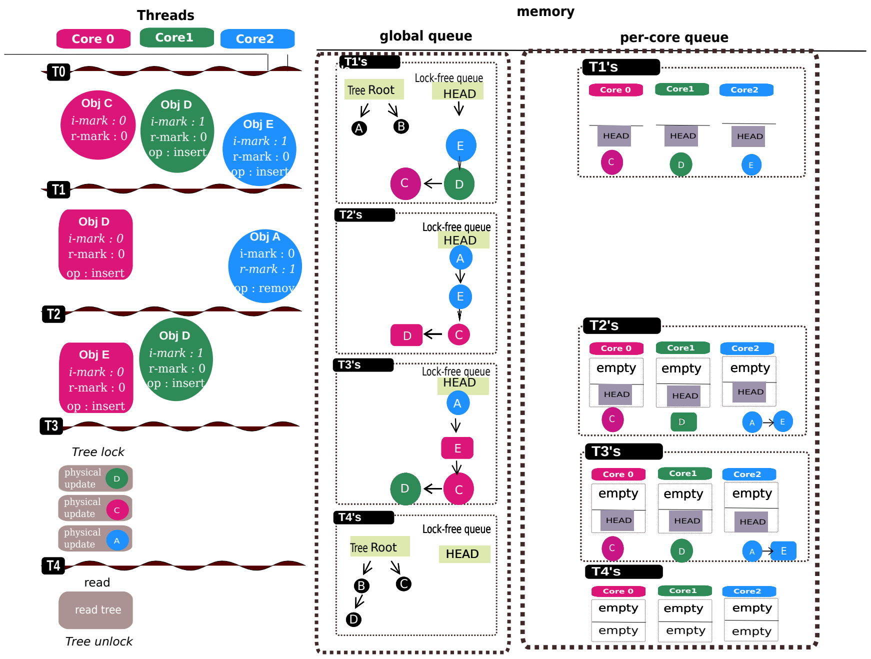


Figure 2: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

A. Experimental setup

B. AIM7

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and arithmetic operation. To minimize IO bottlenecks, the workload was executed with tmpfs filesystems, each of which is 10 GB. To increase the number of users during our experiment and show the results at the peak user numbers, we used the crossover.

The results for AIM7-multiuser are shown in Figure 7, and the results show the throughput of AIM7-multiuser with four different settings. Up to 60 core, the stock Linux scales linearly while serialized updates in Linux kernel become bottlenecks. However, up to 120core, unordered harris list and our LDU scale well because these workloads can run concurrently updates and can reduce the locking overheads due to reader-writer semaphores(`anon_vma`, `file`). The combination of LDU with unordered harris list has best performance and scalability outperforming stock Linux by 1.7x and unordered harris list by 1.1x. While the unordered harris list has 19% idle time(see Table **??**), stock Linux has 51% idle time waiting to acquire both `anon_vma's rwsem` and `file's i_mmap_rwsem`. We can notice that although LDU has 23% idle time, the throughput is higher than unordered harris list. In this benchmark, the ordered harris list has the lowest performance and scalability because their `CAS` fails frequently.

C. Exim

To measure the performance of Exim, shown in Figure 8, we used default value of MOSBENCH to use tmpfs for spool files, log files, and user mail files. Clients run on the same machine and each client sends to a different user to prevent contention on user mail file. The Exim was bottlenecked by per-directory locks protecting file creation in the spool directories and by forks performed on different cores [1]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

Results shown in Figure 8 show that Exim scales well for all methods up to 60 core but not for higher core counts. The stock Linux shows performance degradation for more than 60 core. Both unordered harris list and our LDU do not suffer from performance loss because they do not acquire the `anon_vma` semaphore and `i_mmap` semaphore in fork. LDU performs better due to the fact that it uses both update-side absorbing and lock-less list, outperforming stock Linux by 1.6x and unordered harris list by 1.1x. Even though we applied scalable solution, Exim shows limitation on scalability improvement since the main bottleneck is per-directory lock contention on spool directories. The unordered harris list has 31% idle time, whereas LDU has 37% idle time due to the their efficient concurrent updates.

```
bool ldu_logical_insert(struct object_struct *obj,
    void *head)
{
  ...
  // Phase 1 : update-side removing logs
  // atomic swap due to synchronize update's logs
  if(!xchg(&obj->remove->mark, 0)){
    BUG_ON(obj->insert->mark);
    insert->mark = 1;
    // Phase 2 : reuse garbage log
    if(!test_and_set_bit(LDU_INSERT,
        &obj->ldu.used)){
      // Phase 3 : insert log to queue
      //...save argument and operation
      ldu_insert_queue(root, insert);
    }
  }
  ...
}

bool ldu_logical_remove(struct object_struct *obj,
    void *head)
{
  ...

  // Phase 1 : update-side removing logs
  // atomic swap due to synchronize update's logs
  if(!xchg(&obj->insert->mark, 0)){
    BUG_ON(obj->remove->mark);
    remove->mark = 1;
    // Phase 2 : reuse garbage log
    if(!test_and_set_bit(LDU_REMOVE,
        &obj->ldu.used)){
      // Phase 3 : insert log to queue
      //...save argument and operation
      ldu_insert_queue(root, insert);
    }
  }
  ...
}
```

Figure 3: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

D. Lmbench

lmbench has various workloads including process creation workload(fork, exec, sh -c, exit). This workload is used to measure the basic process primitives such as creating a new process, running a different program, and context switching. We configured process create workload to enable the parallelism option which specifies the number of benchmark processes to run in parallel [14]; we used 100 processes.

The results for lmbench are shown in Figure 10, and the results show the execution times of the fork microbenchmark in lmbench with four different methods. Three methods outperform stock Linux by 2.2x at 120 cores;however, before 30 core, two harris list have lower performance due to their execution overheads. While stock Linux has 90% idle time, other methods have approximately 50% idle time since

```
void synchronize_ldu(struct obj_root *root)
{
  ...

  //atomic remove first, lock-less list
  entry = xchg(&head->first, NULL);

  //iterate all logs
  llist_for_each_entry(dnode, entry, ll_node) {
    //get log's arguments
    ...
    //atomic swap due to update-side removing
    if (xchg(&dnode->mark, 0))
      ldu_physical_update(dnode->op_num, arg,
          ACCESS_ONCE(dnode->root));
    clear_bit(dnode->op_num, &vma->dnode.used);
    // one more check due to reuse garbage log
    if (xchg(&dnode->mark, 0))
      ldu_physical_update(dnode->op_num, arg,
          ACCESS_ONCE(dnode->root));
  }
}
```

Figure 4: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

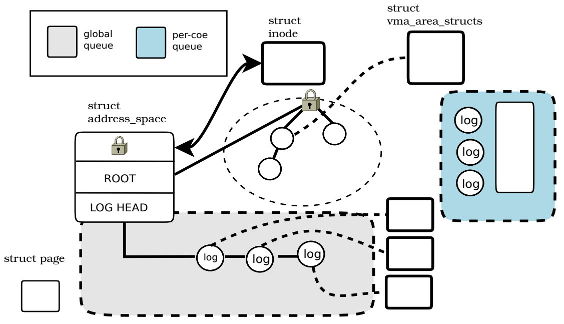


Figure 5: An example of applying the LDU to file reverse mapping.

stock Linux waits to acquire reverse mapping locks such as `anon_vma's rwsem` and `mapping's i_mmap_rwsem`.

E. Updates ratio

VII. Discussion

VIII. Related work

Operating system scalability. [15]In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks [30] [31]. Shared address spaces in multithreaded applications easily become scalability bottlenecks since kernel operations including `mmap` and `munmap` system calls and `page faults` handling require per-process locks for synchronization. Multithreaded application, for example, can become bottleneck by kernel operations on their shared address space, whose operations are the `mmap` and `munmap`

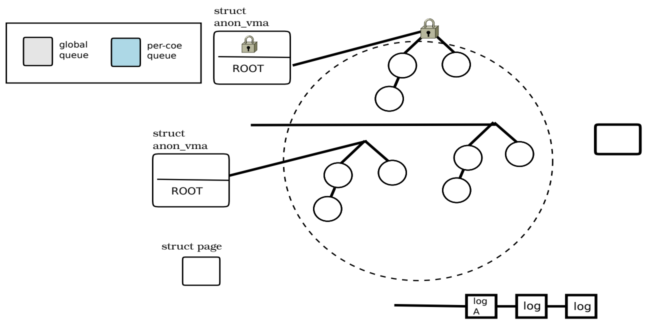


Figure 6: An example of applying the LDU to file reverse mapping.

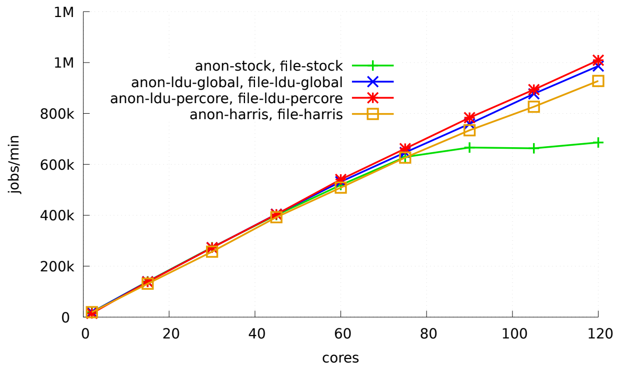


Figure 7: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

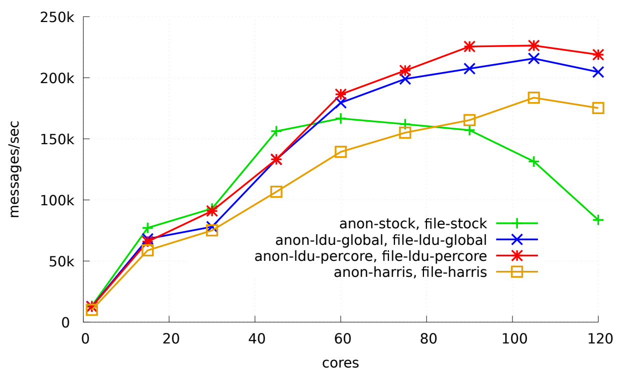


Figure 8: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

system calls and `page faults`. These operations are synchronized by a single per-process lock. BonsaiVM [32] solved this address space problem by using the RCU; RadixVM [33] created a new VM using refcache and radix tree, which enable `munmap`, `mmap`, and `page fault` on non-overlapping memory regions to scale perfectly. Alternatively, to avoid contention caused by shared address space locking, system programmers change their multithreaded applications to use processes [1].

Scalabe data structure. [4]Though sufficient level of performance scalability has been achieved for reader intensive operations through RCU and Hazard pointer, solutions to scalability for update-heavy operations has not been satisfiable. A recent paper by Arbel and Attiya [6] shows a new design of concurrent search tree called the Citrus tree. The Citrus tree combines RCU and fine-grained locks, and it supports concurrent write operations that traverse the search tree by using RCU concurrently. When increasing the update rate, Citrus tree still suffers from bottlenecks. RLU [3] presents a new synchronization mechanism that allows unsynchronized sequences of reads to execute concurrently with updates. In high update rate, Oplog can achieve substantially multi-core scaling for update-heavy data structures. Our work focus on update-heavy data structures and uses non-blocking method to store the operation log instead of per-core processing.

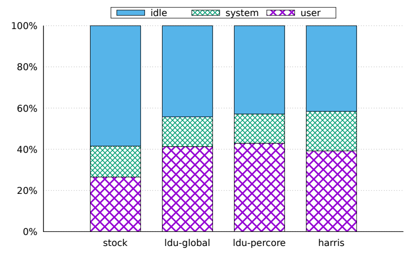
Scalable lock. [34] [16] [17]One method for the concurrent update is using the non-blocking algorithms [28] [35] [36], which are based on CAS.

MCS [37], a scalable exclusive lock, is used in the Linux kernel [38]. to avoid unfairness at high contention levels, so this scalable exclusive lock can be used for fine grained locking in Linux. However, in MCS, since only one thread may hold the lock at a time, it can cause low scalability in case of long critical regions. Reader-writer lock [39] allows either number of readers to execute concurrently or single writer to execute. Thus, readers-writer locks allow better scalability in case of read-mostly objects.

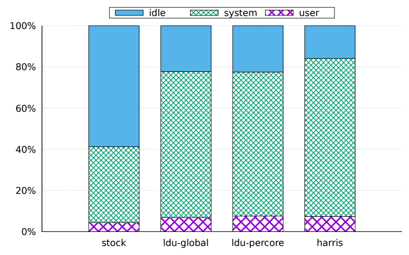
In read-mostly data structures, RCU [40] can be quite useful since it allows read operations to proceed without read locks, and delays freeing of data structures to avoid races. One drawback is that as the update rate increases, their performance and scalability decrease due to a single writer and their synchronization function. Consequently, scalable exclusive lock, reader-writer lock and RCU require serialization for updates and thus show significant limitation on scalability.

IX. Conclusion

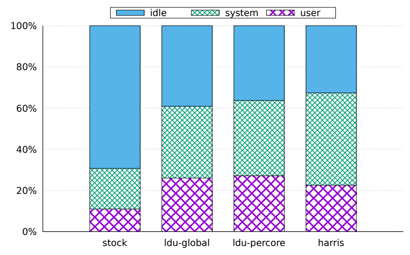
We propose a concurrent update algorithm, LDU, for update-heavy data structures scalable for many-core systems. To achieve the scalability during process spawning, we applied deferred log processing with global log queue and update-side absorbing to Linux reverse mapping. Evaluation results using the AIM7, Exim and lmbench reveal that LDU shows better performance up to 2.2 times compared to existing solutions. LDU is implemented on to Linux kernel



(a) AIM7 - 120core



(b) Exim - 120core



(c) Lmbench - 120core

Figure 9: Read-write ratio from 50:50 to 1:99 percent

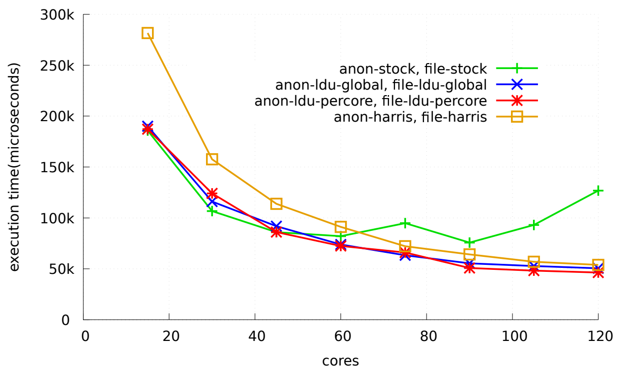


Figure 10: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

4.5 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 9th *USENIX Symposium on Operating System Design and Implementation*, Vancouver, BC, Canada, October 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)*. Denver, CO: USENIX Association, Jun. 2016, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[3] 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, New York, NY, USA, 2015, pp. 168–183.

[4] 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. New York, NY, USA: ACM, 2015, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676963

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

[6] 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, New York, NY, USA, 2014, pp. 196–205.

[7] O. Shalev and N. Shavit, "Predictive log-synchronization," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 305–315, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1218063.1217965

[8] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 355–364. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810540

[9] S. Boyd-Wickizer, "Optimizing communications bottlenecks in multiprocessor operating systems kernels," in *PhD thesis, Massachusetts Institute of Technology*, 2013.

[10] P. McKenney, "Msome more details on read-log-update," 2016, https://lwn.net/Articles/667720/.

[11] "Aim benchmarks," http://sourceforge.net/projects/aimbench.

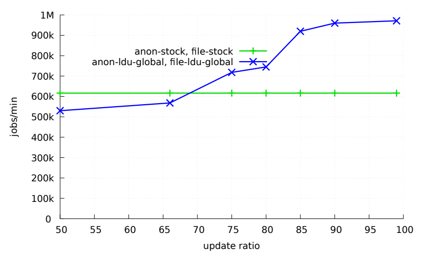
[12] "Exim internet mailer," 2015, http://www.exim.org/.

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

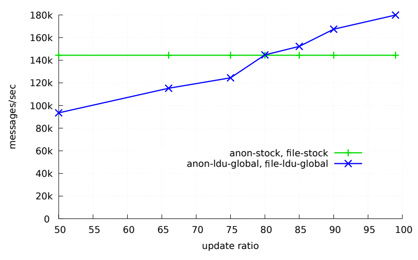
[14] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[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," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 10:1–10:47, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699681

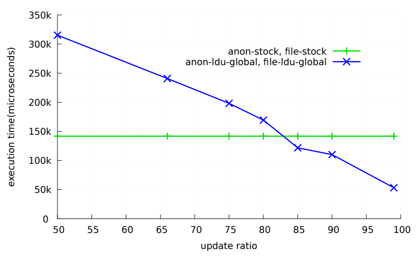
[16] D. Bueso, "Scalability techniques for practical synchronization primitives," *Commun. ACM*, vol. 58, no. 1, pp. 66–74, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2687882



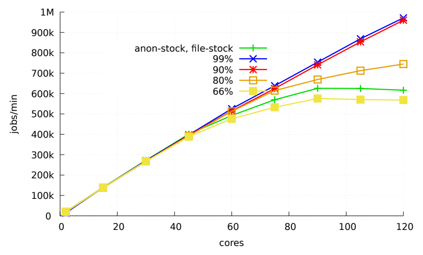
(a) AIM7 - 120core



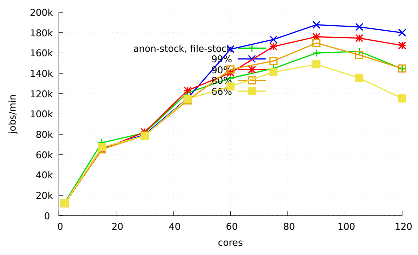
(b) Exim - 120core



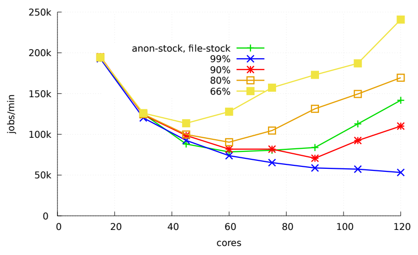
(c) Lmbench - 120core



(d) AIM7 - scalability



(e) Exim - scalability



(f) Lmbench - scalability

Figure 11: Read-write ratio from 50:50 to 1:99 percent

[17] 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.

[18] C. Rohland, "Tmpfs is a file system which keeps all files in virtual memory," 2001, git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/tmpfs.txt.

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

[20] D. H. Tim Chen, Andi Kleen, "Linux scalability issues," in *Linux Plumbers Conference, September*, 2013.

[21] 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, New York, NY, USA, 2015, pp. 631–644.

[22] S. Al Bahra, "Nonblocking algorithms and scalable multicore programming," *Commun. ACM*, vol. 56, no. 7, pp. 50–61, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2483852.2483866

[23] E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*, 2013.

[24] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical non-blocking unordered lists," in *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*, 2013.

[25] H. Ying, "Lock-less list," 2011, https://lwn.net/Articles/423366/.

[26] J. Corbet, "The object-based reverse-mapping vm," 2003, https://lwn.net/Articles/23732/.

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

[28] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01, London, UK, UK, 2001, pp. 300–314.

[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, New York, NY, USA, 2015, pp. 1–10.

[30] 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. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855745

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

[32] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, London, UK, February 2012, pp. 199–210.

[33] ——, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, New York, NY, USA, 2013, pp. 211–224.

[34] T. Wang, M. Chabbi, and H. Kimura, “Be my guest: Mcs lock now welcomes guests,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’16. New York, NY, USA: ACM, 2016, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851160

[35] 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.

[36] 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, New York, NY, USA, 2012, pp. 309–310.

[37] J. M. Mellor-Crummey and M. L. Scott, “Scalable reader-writer synchronization for shared-memory multiprocessors,” in *Proceedings of the Third PPOPP*, Williamsburg, VA, April 1991, pp. 106–113.

[38] J. Corbet., *MCS locks and qspinlocks*, 2014.

[39] P. J. Courtois, F. Heymans, and D. L. Parnas, “Concurrent control with “readers” and “writers”,” *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, October 1971.

[40] P. E. McKenney and J. D. Slingwine, “Read-copy update: Using execution history to solve concurrency problems,” in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.