LDU: A LIGHTWEIGHT CONCURRENT UPDATE METHOD WITH DEFERRED PROCESSING FOR LINUX KERNEL SCALABILITY

Joohyun Kyong and Sung-Soo Lim
School of Computer Science
Kookmin University
Seoul, Korea
email: joohyun0115@gmail.com, sslim@kookmin.ac.kr

ABSTRACT
We propose a novel light weight concurrent update 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 3.19 and evaluated using representative benchmark programs. Our evaluation reveals that the Linux kernel with LDU shows performance improvement by ranging from 1.2x through 2.2x on a 120 core system.

KEY WORDS
Scalability, Operating System, Linux, Concurrent Update

1 Introduction

With the drastic increase of CPU core counts in various high-end server systems, achieving performance scalability of operating systems running on such many-core systems has been an important issue in research communities. Linux has been naturally considered as a major target for the scalability improvement and a number of accomplishments are published. Early results include RCU [16] and hazard pointer [19] to improve scalability for read-most data structures in the Linux kernel. Though the early results show certain level of improvement in scalability, it turned out that more significant portion of scalability limitation of Linux kernel is due to lock contention in update-heavy global data structures including file reverse mappings and anonymous reverse mappings during spawning child processes [2] [21]. Such update-heavy data structures cause serialization of the update operations leading to severe performance degradation.

To solve this problem, an existing approach is to make the update-heavy data structures as non-blocking [13] based on *compare-and-swap*(CAS). Introducing non-blocking data structures eliminates the update serialization problem during process spawning, but incurs additional issues due to inter-core communication bottlenecks and cache coherence system's write serialization [4]. To overcome the issues caused by cache coherence system, S. Boyd-Wickizer et al. proposed Oplog [4] where logs update operations with time stamps and actual updates are performed later when the updated data need to be read. While Oplog nicely solves the update serialization problem without any cache coherence-related overheads, the merging of the update logs recorded in multiple per-core data structures considering time stamps further causes performance overheads resulting in limited scalability improvement [15].

This paper proposes a novel concurrent update method, LDU, applicable to Linux reverse mapping solving the problems mentioned above: the overheads caused by inter-core communication bottlenecks and per-core log management with time stamps.

The LDU is similar to Oplog in that it defers the actual update operations as late as possible to reduce serialization problems, but it uses a light weight global queue with non-blocking synchronization for update logs and eliminates time stamps required for per-core log management. In addition, to optimize the log management and minimize the traversal overheads during reading, LDU applies update-side absorbing algorithm based on atomic marking and thus efficiently find the operations to be canceled. The evaluation of the proposed LDU on Linux kernel 3.19.rc4 running on a 120 core system reveals that the execution times could be improved by 1.7x, 1.6x, and 2.2x for a fork-intensive workload- AIM7 [1], Exim from MOSBENCH [5], and lmbench [17], respectively.

This paper is organized as follows. Section 2 summarizes related works and compare our contributions to previous works. Section 3 describes the design of the LDU algorithm and Section 4 explains how to apply to Linux kernel. section 5 explains our implementations in Linux and Section 6 shows the results of the experimental evaluation. Finally, section 7 concludes the paper.

2 Background and related work

2.1 Linux Scalability

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. BonsaiVM [6] solved this address space problem by using the RCU; RadixVM [7] 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 [5].

Multi-processing environment suffers from scalability bottlenecks due to the well-known fork scalability problem [2] [21]. When Linux spawns child process, the Linux substantially performs locks because of protecting the reverse mapping data structure naturally causing bottlenecks. Oplog [4], which is an important basis of our approach, solves this problem using time-stamp log and per-core processing.

2.2 Locking

MCS [18], a scalable exclusive lock, is used in the Linux kernel [8]. 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 [9] 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 [16] 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.

2.3 Non-Blocking algorithm

One method for the concurrent update is using the non-blocking algorithms [13] [11] [**?**], which are based on CAS. In non-blocking algorithms, each core tries to read the values of shared data structures from its local location, but has possibility of reading obsolete values. CAS is performed at the time of reading values that are not the current values and CAS fails and requires retrials sometimes when the values have been overwritten. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [4]. Moreover, none of the non-blocking algorithms implements an iterator, whose data structure just consists of the insert, delete and contains operations [20]. The Linux, however, commonly uses the iteration to read, so when applying non-blocking algorithms to the Linux, they may meet this iteration problem. Petrank [20] solved this problem by using a consistent snapshot of the data structure; this method, however, may require a lot of effort to apply its sophisticated algorithms to Linux. For evaluation purposes, we implemented Harris linked list [13] to Linux, and we sometimes have failure where reading the pointer that had been deleted by updater concurrently result of the problem of the iteration.

Linux kernel uses lock-less list("lock-less NULL terminated single list") that are widely used in the Linux kernel to improve scalability. In order to delete operation for multiple consumer, the existing algorithms traverse the list from beginning or their optimized point. On the other hand, lock-less list inserts node at the first of the list, so when the CAS operation fails, they will minimally traverse from the head node. Although lock-less list uses a non blocking method, they retries minimally and thus they can significantly reduce inter-core communication bottleneck and the repeated loop bottleneck. Our proposed method uses this feature in case of inserting the operation log.

2.4 Concurrent Update

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 [3] 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 [14] 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.

3 LDU Algorithm

This section describes the LDU algorithm, a lightweight concurrent update for update-heavy data structures based on deferred updates. Challenges to designing a deferred update mechanism includes performing concurrent update with minimal cache line transfers allowing parallel updates. At each update operation, LDU records this update operation log to lock-less list. Before the read operation, LDU applies the updates log in chronological order. In order to deferred update, LDU divide the update operation into *logical update* and *physical update*. The *logical update* inserts logs into the lock-less list and carries out update side absorbing; on the other hand, the *physical update* executes these operations that are minimized by the update side absorbing.

3.1 Approach

LDU's scheme for concurrent update is proposed to overcome limitations of Linux kernel where both insert and re-

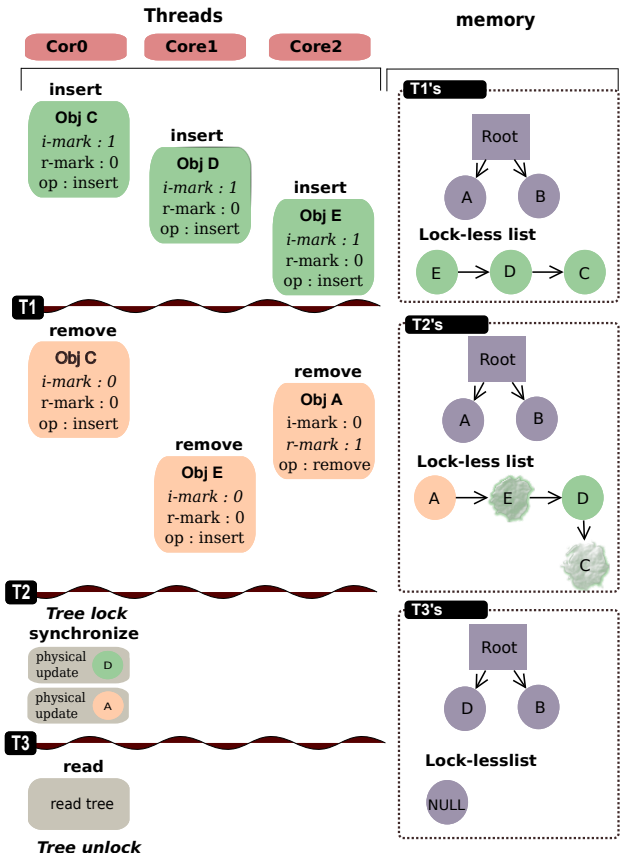

Figure 1: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

move operations must not be invoked concurrently for the same object, but reads can be concurrently invoked with update. LDU borrows ideas from Oplog's deferred processing and Harris' marking scheme.

One important algorithm in our proposed novel concurrent update scheme is update-side absorbing operation that cancels duplicated operations for optimizations. A new remove operation, for example, may cancel an existing insert operation with regard to same object, so reader can eventually reads consistent data. Even though the Oplog's absorbing operation is invoked by read, LDU's absorbing operation is fully invoked by update, so read-side performance is enhanced.

The basic principle of update-side absorbing is that update uses atomic marking operation for the object's mark field, which allows previous operation to cancel. For instance, if a new remove operation occurs after insert operation of the same object, LDU does not store this operation in the lock-less list; instead, it changes the insert mark field to zero using the CAS. This mark is checked later when reading operation occurs and the operation log maintained in the lock-less list is applied to original data structure atomically.

Figure 1 gives an example of deferred update with six update operations and one read operation. The data structure for *physical update* is a tree, and initial values in the tree are node `A` and `B`. In contrast, the data structure for *logical update* is lock-less list. In the top figure, `Core0`, `Core1` and `Core2` perform the logical insert operation to nodes `C`, `D` and `E`, respectively. The logical inserts set the insert mark, and they then insert their nodes into lock-less list. In this case, none of the lock is needed because LDU uses the lock-less list;all threads can execute the update concurrently. At `T1`, the tree contains node `A` and `B` and the lock-less list contains node `E`, `D` and `C`. When removing the node `C`, the node `C`, whose mark field was marked by insert, atomically cleans up the insert marked field. At `T2`, the lock-less list contains nodes `A`, `E`, `D`, and `C`, and the marking field is zero for nodes `E` and `C`. Before running the `synchronize` function, they need to lock the original tree's lock using the exclusive lock in order to protect the tree's operation. The `synchronize` migrates from lock-less list node to tree node, each of which is the marked node, so nodes `A` and `D` are migrated. Finally, the tree contains nodes `D` and `B`, so the reader can read eventually consistent data.

One notable difference between Oplog and LDU is that LDU uses a light weight global queue with non-blocking synchronization for update logs and eliminates time stamps while Oplog is dependent on per-core logs with time stamps. By eliminating the global time stamps(hardware-dependent feature), LDU is not dependent on hardware feature. Furthermore, to optimize the log management and minimize the traversal overheads during reading, LDU applies efficient update-side absorbing algorithm instead of read-side absorbing algorithm.

3.2 logical update

The pseudo code for LDU's *logical update* is given in figure 2. The `logical_insert`, the concurrent update function, checks whether this object already has been removed by `logical_remove`. If this object has been removed, `logical_insert` initializes the marking field and then they return, which is fastpath. The marking field needs synchronization because this field in the *logical update* is shared with the *physical update*, so the CAS operation is needed. When the marking field has been initialized, they set the marking field, then they check whether or not this node already has been inserted in lock-less list. If the node does not exist in lock-less list, then they insert the node into lock-less list.

3.3 Physical update

The pseudo code for LDU's *physical update* is given in Figure 3. First, they check whether lock-less list is an empty list or not, then they iterate the lock-less list. If the marking field has been set, they execute migration from lock-less to

```
function logical_insert(obj, root):
  If CAS(obj.del_node.mark, 1, 0) ≠ 1:
    obj.add_node.mark ← 1
    If test_and_set_bit(OP_INSERT, obj.exist) ≠ true:
      set_bit(OP_INSERT, obj.used):
      obj.add_node.op ← OP_INSERT
      obj.add_node.key ← obj
      obj.add_node.root ← root
      add_lock_less_list(obj.add_node)


function logical_remove(obj, root):
  If CAS(obj.add_node.mark, 1, 0) ≠ 1:
    obj.del_node.mark ← 1
    If test_and_set_bit(OP_REMOVE, obj.exist) ≠ true:
      set_bit(OP_REMOVE, obj.used):
      obj.del_node.op ← OP_REMOVE
      obj.del_node.key ← obj
      obj.del_node.root ← root
      add_lock_less_list(obj.del_node)
```

Figure 2: 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.

original data structure. Because the marking field in *physical update* is shared with *logical update*, the CAS operation is needed. They initialize the used field, which needs to protect the object from freed through destructor. The programmer must acquire locks on the `synchronize_ldu` function, which migrates log to original data structure. Finally, the `physical_update` executes original functions by using the operation log.

4 Concurrent update for Linux kernel

In this section, we describe how to apply our concurrent update based on deferred update method to Linux. The Linux fork is associate with an anonymous page and a file page. When many processes are simultaneously created in Linux, these two reverse mapping can become bottlenecks since their data structures are shared between processes. Figure 4 shows the scalability problem in case of the fork-intensive workload that simultaneously creates many processes. Up to 60 core, the stock Linux scales linearly, then creating the reverse mapping becomes the bottleneck because their interval trees are protected by locks. Therefore, fork-intensive workload can pose a scalability bottleneck due to the update-heavy data structures [4] [2] [21].

Figure 5 gives an example of applying the LDU to file reverse mapping and shows relationship between interval trees and lock-less lists. An interval tree contains two `virtual memory area`(`VMA`) nodes; on the other hand, the lock-less list contains the right `VMA` as shown in Figure 5. It means that the right `VMA` has been deleted, and the synchronization has not been invoked.

In order to using the LDU, the data structures involved in the head(`address_space`) or the node(`vm_area_structure`) can be modified with LDU's structure as shown in Figure 5. In addition, programmer must replace *physical update* with *logical update* to eliminate the lock. Before the corresponding readers need to be read, LDU must call synchronize function to keep the consistency.

```
function synchronize_ldu(obj, head):
  If (head.first = NULL):
    return;
  entry ← xchg(head.first, NULL);
  for each list node:
    obj ← node.key
    clear_bit(node.op, obj.exist)
    If CAS(node.mark, 1, 0) = 1:
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)

function physical_update(op, obj, root):
  If op = OP_INSERT :
    call real insert function(obj, root)
  Else If op = OP_REMOVE :
    call real remove function(obj, root)
```

Figure 3: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

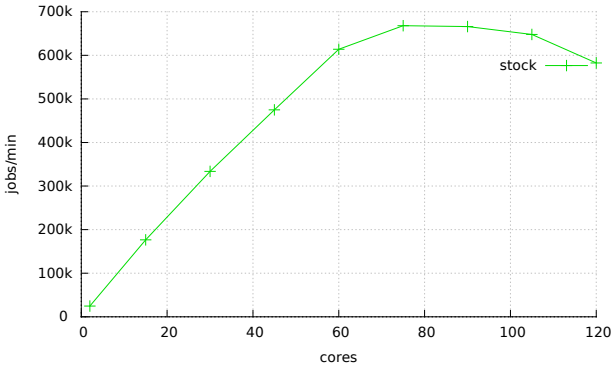

Figure 4: Scalability of AIM7 multiuser. This workload simultaneously create many processes. Up to 60 core, the stock Linux scale linearly, then they flattens out.

5 Implementation

We implemented the new deferred update algorithm in Linux 3.19.rc4 kernel, and our modified Linux is available as open source. LDU's scheme is based on deferred pro-

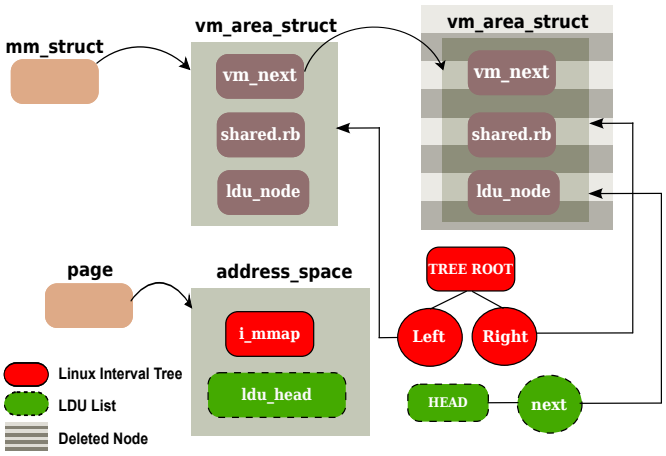

Figure 5: An example of applying the LDU to file reverse mapping.

cessing, so it needs a garbage collector for delayed free. In order to implement the garbage collector, we use the lock-less list and a periodic timer(1 sec) in the Linux.

We compare our LDU implementation to a concurrent non-blocking Harris linked list [13];therefore, we implement the Harris linked list to Linux kernel. The code refers from sysnchrobench [12] and ASCYLIB [10], and we convert their linked list to Linux kernel style. Because both synchrobench and ASCYLIB leak memory, we implement additional garbage collector for the Linux kernel using Linux's work queues and lock-less list.

In order to further improve performance, we move their ordered list to unordered list. A feature of the Harris linked list is all the nodes are ordered by their key. Zhang [22] implements a lock-free unordered list algorithm, whose list is each insert and remove operation appends an intermediate node at the head of the list;these approach is practically hard to implement. Indeed, Linux does not require contains operation because the Linux data structures such as list, tree and hash table not depended on search key;they depend on their unique object. This feature can eliminates the ordered list in Harris linked list. Therefore, we perform each insert operation appends an intermediate node at the first node of the list;on the other hand, each remove operation searches from head to their node.

To the scalability of fork, the reverse mapping's lock contention should be eliminated not only from file reverse mapping but also from anonymous reverse mapping. The structure of file reverse mapping is simplified relatively to the structure of anonymous mapping because the anonymous reverse mapping is entangled by their global object(`anon_vma`) and their chain(`anon_vma_chain`);therefore, we only apply LDU to file reverse mapping.

6 Evaluation and Discussion

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?

6.1 Experimental setup

To evaluate the performance of LDU, we use well-known three benchmarks:AIM7 Linux scalability benchmark, Exim email server in MOSBENCH and lmbench. We selected these three benchmarks because they are fork-intensive workloads and exhibit high reverse mapping lock contentions. Moreover, AIM7 benchmark has widely been used in practical area not only for testing the Linux but also for improving the scalability. To evaluate LDU for real world applications, we use Exim which is the most popular email server. A micro benchmark, Lmbench, has been selected to focus on Linux fork operation-intensive fine grained evaluations.

In order to evaluate Linux scalability, we used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used ordered Harris lock-free list while we apply unordered Harris lock-free list for the third setting (see section 5). Finally, we used combination of unordered Harris lock-free list for anonymous mapping and our LDU for file mapping. Since we cannot obtain detailed implementation of Oplog, we could not include comparison between LDU and Oplog in this paper.

We ran the three benchmarks on Linux 3.19.rc4 with stock Linux with the automatic NUMA balancing feature disabled because the Harris linked list has the iteration issue [20]. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

6.2 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 6, 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

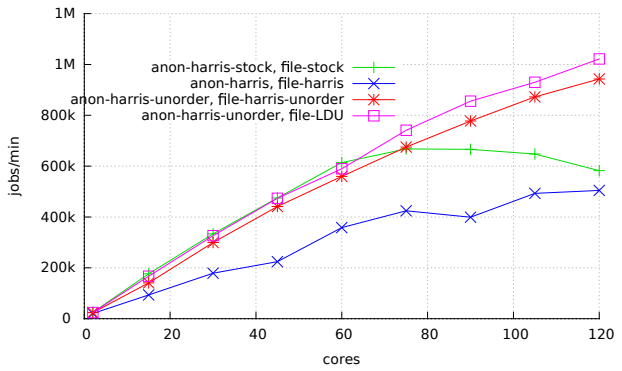

Figure 6: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

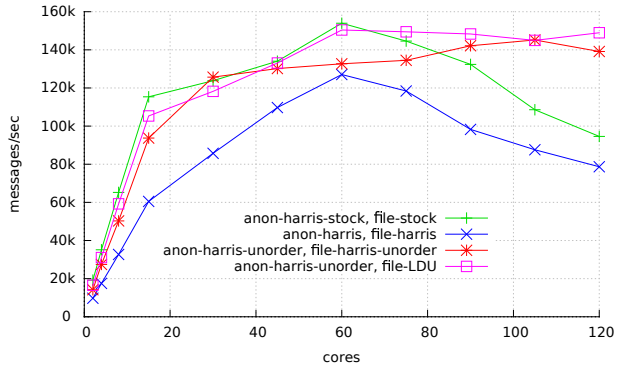

Figure 7: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

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 1), 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.

6.3 Exim

To measure the performance of Exim, shown in Figure 7, we used default value of MOSBENCH to use tmpfs for spool files, log files, and user mail files. Clients run on

AIM7	user	sys	idle
Stock(anon, file)	2487 s	1993 s	4647 s(51%)
H(anon, file)	1123 s	3631 s	2186 s(31%)
H-unorder(anon, flie)	3630 s	2511 s	1466 s(19%)
H-unorder(anon), L(file)	3630 s	1903 s	1662 s(23%)

EXIM	user	sys	idle
Stock(anon, file)	41 s	499 s	1260 s(70%)
H(anon, file)	47 s	628 s	1124 s(62%)
H-unorder(anon, file)	112 s	1128 s	559 s(31%)
H-unorder(anon), L(file)	87 s	1055 s	657 s(37%)

lmbench	user	sys	idle
Stock(anon, file)	11 s	208 s	2158 s(91%)
H(anon, file)	11 s	312 s	367 s(53%)
H-unorder(anon, file)	11 s	292 s	315 s(51%)
H-unorder(anon), L(file)	12 s	347 s	349 s(49%)

Table 1: Comparison of user, system and idle time at 120 cores.

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 [5]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

Results shown in Figure 7 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.

6.4 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 [17]; we used 100 processes.

The results for lmbench are shown in Figure 8, and the results show the execution times of the fork microbenchmark in lmbench with four different methods. Three meth-

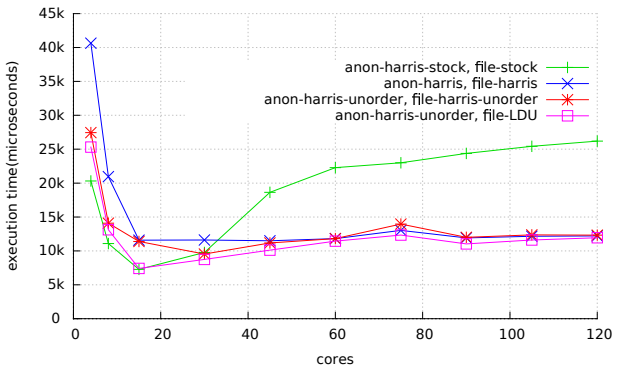

Figure 8: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

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

7 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 3.19 and available as open-source from `https://github.com/KMU-embedded/scalablelinux`.

8 Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (14-824-09-011, "Research Project on High Performance and Scalable Manycore Operating System")

References

[1] Aim benchmarks. `http://sourceforge.net/projects/aimbench`.

[2] Tim Chen Andi Kleen. Scaling problems in fork. 2011.

[3] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014.

[4] Silas Boyd-Wickizer. Optimizing communications bottlenecks in multiprocessor operating systems kernels. 2014.

[5] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 199–210, London, UK, February 2012.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013.

[8] J. Corbet. Mcs locks and qspinlocks, 2014. `https://lwn.net/Articles/590243/`.

[9] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[10] Tudor David, Rachid Guerraoui, and Vasileios 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*, ASPLOS '15, pages 631–644, New York, NY, USA, 2015.

[11] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, 2004.

[12] Vincent 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*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015.

[13] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001.

[14] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015.

[15] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2011. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`.

[16] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[17] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[18] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP*, pages 106–113, Williamsburg, VA, April 1991.

[19] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. In *IEEE Transactions on Parallel and Distributed Systems*, volume 15, pages 491–504, June 2004.

[20] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*. 2013.

[21] Dave Hansen Tim Chen, Andi Kleen. Linux scalability issues. In *Linux Plumbers Conference, September*, 2013.

[22] Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael Spear. Practical non-blocking unordered lists. In *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*. 2013.