A Lightweight Log-based Deferred Concurrent Updates for Linux Kenrel Scalability

Your N. Here
Your Institution

Second Name

Name
Name Institution

Abstract

1 Introduction

2 Background and Problem

3 LDU Design

LDU This section explains these algorithmic design aspects of LDU.

3.1 Log-based Concurrent updates

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

3.2 Approach

For update-heavy data structure LDU update before read operation;for example, it similar to CoW(Copy On Write) method because acture update as long as possible late.

3.3 LDU example

3.3.1 GLDU example

Figure 2 gives an example of deferred update with six update operations and one read operation. In this figure, execution flows from top to bottom. 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.

3.3.2 PLDU example

Figure 2 gives an example of deferred update with six update operations and one read operation. In this figure, execution flows from top to bottom. 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.

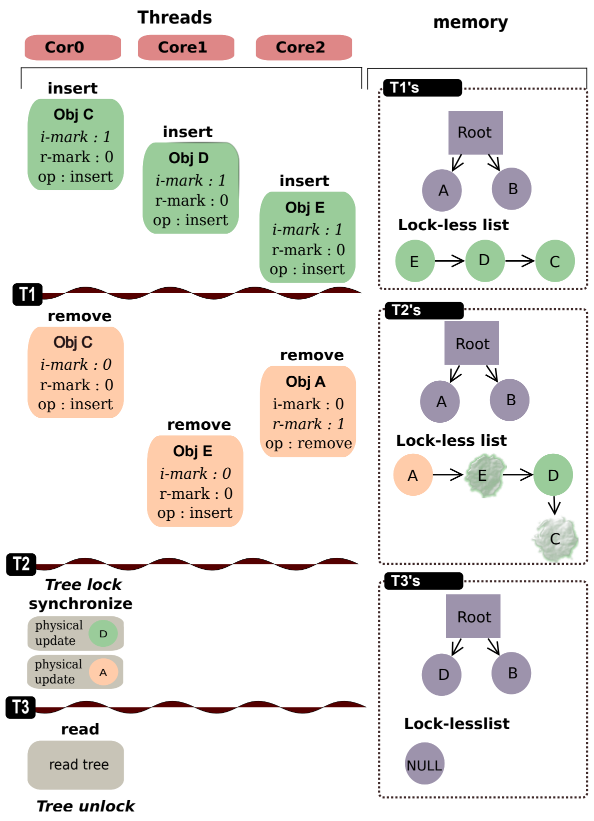

Figure 1: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

3.4 GLDU vs. PLDU

3.5 The LDU Algorithm

3.5.1 GLDU logical update

The pseudo code for LDU's *logical update* is given in figure 5. 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.

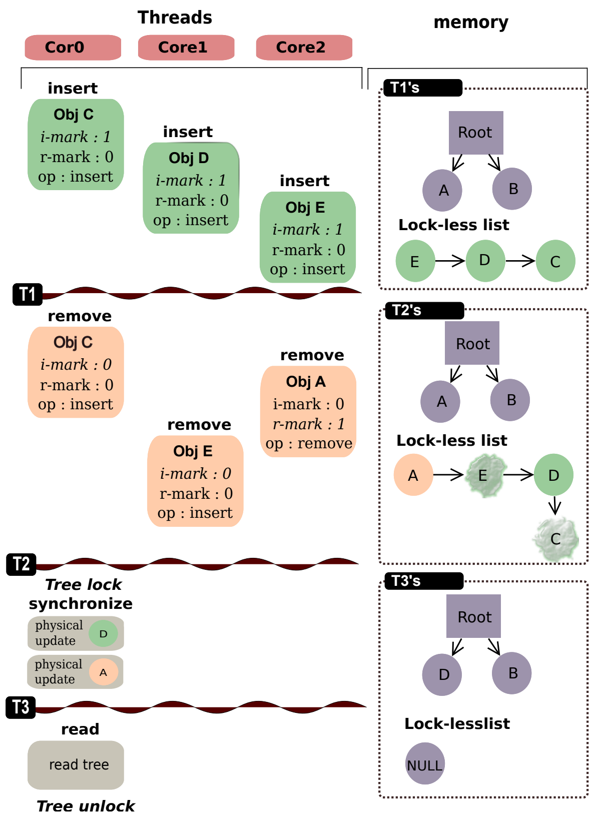

Figure 2: LDU example showing six update operations and one read operation. The execution flows from top to bottom. Memory represents original data structure and logging queue at T1, T2 and T3, respectively.

3.5.2 GLDU Physical update

The pseudo code for LDU's *physical update* is given in Figure 6. 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 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.

3.5.3 The PLDU Algorithm

3.5.4 PLDU logical update

* By default OpLog executes logged updates in temporal order. For example, consider a linked list. If a process

```

bool ldu_logical_insert(struct vm_area_struct *vma,
    struct address_space *mapping)
{
  struct ldu_node *add_dnode = &vma->dnode.node[0];
  struct ldu_node *del_dnode = &vma->dnode.node[1];

  if (atomic_cmpxchg(&del_dnode->mark, 1, 0) != 1) {
    atomic_set(&add_dnode->mark, 1);
    if (!test_and_set_bit(LDU_OP_ADD, &vma->dnode.used)) {
      add_dnode->op_num = LDU_OP_ADD;
      add_dnode->key = vma;
      add_dnode->root = &mapping->i_mmap;
      i_mmap_ldu_logical_update(mapping, add_dnode);
    }
  }

  return true;
}

bool ldu_logical_remove(struct vm_area_struct *vma,
    struct address_space *mapping)
{
  struct ldu_node *add_dnode = &vma->dnode.node[0];
  struct ldu_node *del_dnode = &vma->dnode.node[1];

  if (atomic_cmpxchg(&add_dnode->mark, 1, 0) != 1) {
    atomic_set(&del_dnode->mark, 1);
    if (!test_and_set_bit(LDU_OP_DEL, &vma->dnode.used)) {
      del_dnode->op_num = LDU_OP_DEL;
      del_dnode->key = vma;
      del_dnode->root = &mapping->i_mmap;
      i_mmap_ldu_logical_update(mapping, del_dnode);
    }
  }

  return true;
}
```

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.

logs an insert operation on one core, migrates to another core, and logs a remove operation, the remove should eventually execute after the insert. * OpLog relies on timestamps from a system-wide synchronized clock to tell it how to order entries in different cores' log. *This ordering ensures linearizaility, making OpLog compatible with existing data structure semantics.

4 Concurrent updates for Linux kernel

4.1 Case study:reverse mapping

4.2 anon vma

* LDU.
  * PLDU.

```

void ldu_physical_update(int op,
    struct vm_area_struct *vma,
    struct rb_root *root)
{
  if (op == LDU_OP_ADD)
    vma_interval_tree_insert(vma, root);
  else
    vma_interval_tree_remove(vma, root);
}

void synchronize_ldu(struct address_space *mapping)
{
  struct llist_node *entry;
  struct ldu_node *dnode, *next;
  struct ldu_head *lduh = &mapping->lduh;

  entry = llist_del_all(&lduh->ll_head);
  llist_for_each_entry_safe(dnode,
      next, entry, ll_node) {
    struct vm_area_struct *vma =
        ACCESS_ONCE(dnode->key);
    if (atomic_cmpxchg(&dnode->mark,
        1, 0) == 1) {
      ldu_physical_update(dnode->op_num,
          vma,
          ACCESS_ONCE(dnode->root));
    }
    clear_bit(dnode->op_num, &vma->dnode.used);
    if (atomic_cmpxchg(&dnode->mark, 1, 0) == 1) {
      ldu_physical_update(dnode->op_num, vma,
          ACCESS_ONCE(dnode->root));
    }
  }
}

void free_work_func(struct work_struct *work)
{
  struct address_space *mapping =
      container_of(work, struct address_space,
      lduh.sync.work);
  struct llist_node *entry;
  struct vm_area_struct *vnode, *vnext;

  i_mmap_lock_write(mapping);
  synchronize_ldu_i_mmap(mapping);
  i_mmap_unlock_write(mapping);

  entry = llist_del_all(&mapping->llclean);
  llist_for_each_entry_safe(vnode,
      vnext, entry, llist) {
    if (!vnode->dnode.used)
      kmem_cache_free(vm_area_cachep,
          vnode);
    else
      llist_add(&vnode->llist,
          &mapping->llclean);
  }
}
```

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.

4.3 file mapping

* LDU.
  * PLDU.

```
bool ldu_logical_update(struct address_space *mapping,
    struct ldu_node *dnode)
{
  struct pldu_deferred_i_mmap *p;
  struct i_mmap_slot *slot;
  struct llist_node *first;
  struct llist_node *entry;
  struct ldu_node *ldu;
  struct address_space *locked_mapping = NULL;

  slot = &get_cpu_var(i_mmap_slot);
  p = &slot->mapping[hash_ptr(mapping, HASH_ORDER)];
  first = READ_ONCE(p->list.first);
  if (first) {
    ldu = llist_entry(first, struct ldu_node, ll_node);
    if (READ_ONCE(ldu->root) != READ_ONCE(dnode->root)) {
      //pr_info("conflict hash table\n");
      locked_mapping = READ_ONCE(ldu->key2);
      entry = llist_del_all(&p->list);
      llist_add(&dnode->ll_node, &p->list);
      put_cpu_var(i_mmap_slot);
      down_write(&locked_mapping->i_mmap_rwsem);
      if (entry) {
        synchronize_ldu_i_mmap_internal(entry);
      }
      up_write(&locked_mapping->i_mmap_rwsem);
      goto out;
    }
  }

  llist_add(&dnode->ll_node, &p->list);
  put_cpu_var(i_mmap_slot);

out:
  return true;
}

bool i_mmap_ldu_logical_insert(
    struct vm_area_struct *vma,
    struct address_space *mapping)
{
  struct ldu_node *add_dnode = &vma->dnode.node[0];
  struct ldu_node *del_dnode = &vma->dnode.node[1];

  if (atomic_cmpxchg(&del_d->mark, 1, 0) != 1) {
    BUG_ON(atomic_read(&add_dnode->mark));
    atomic_set(&add_dnode->mark, 1);
    if (!test_and_set_bit(OP_ADD, &vma->used)) {
      add->op_num = OP_ADD;
      add->key = vma;
      add->key2 = mapping;
      add->root = &mapping->i_mmap;
      ldu_logical_update(mapping, add_dnode);
    }
  }

  return true;
}
```

Figure 5: 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.

```
void ldu_physical_update(int op,
    struct vm_area_struct *vma,
    struct rb_root *root)
{
  if (!root)
    return;

  if (op == LDU_OP_ADD)
    vma_interval_tree_insert(vma, root);
  else
    vma_interval_tree_remove(vma, root);
}

void synchronize_ldu(struct llist_node *entry)
{
  struct ldu_node *dnode;
  struct address_space *mapping;
  struct vm_area_struct *vma;

  llist_for_each_entry(dnode,
      entry, ll_node) {
    vma = READ_ONCE(dnode->key);
    if (atomic_cmpxchg(
        &dnode->mark, 1, 0) == 1) {
      ldu_physical_update(
          dnode->op_num, vma,
          READ_ONCE(dnode->root));

    }
    clear_bit(dnode->op_num,
        &vma->dnode.used);
    if (atomic_cmpxchg(
        &dnode->mark, 1, 0) == 1) {
      ldu_physical_update(
          dnode->op_num, vma,
          READ_ONCE(dnode->root));
    }
  }
}
```

Figure 6: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

5 Implementation

6 Evaluation

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. Finally, we wanted to focus on Linux fork performance and scalability;therefore, we selected lmbench, a micro benchmark.

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 compare our LDU implementation to a concurrent non-blocking Harris linked list [?];therefore, we implement the Harris The code refers from sysnchrobench [?] and ASCYLIB [?], 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 [?] 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.

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 [?]. 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.

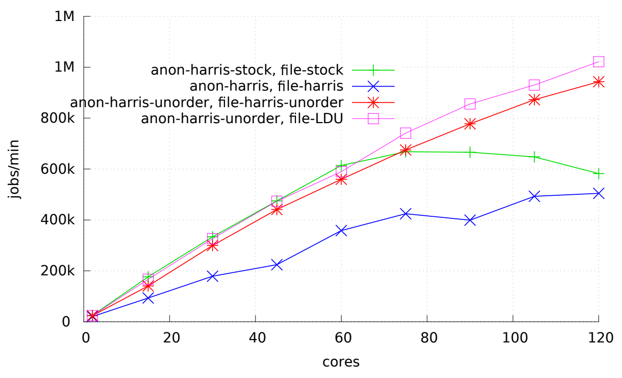

Figure 7: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

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 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 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 [?]. 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

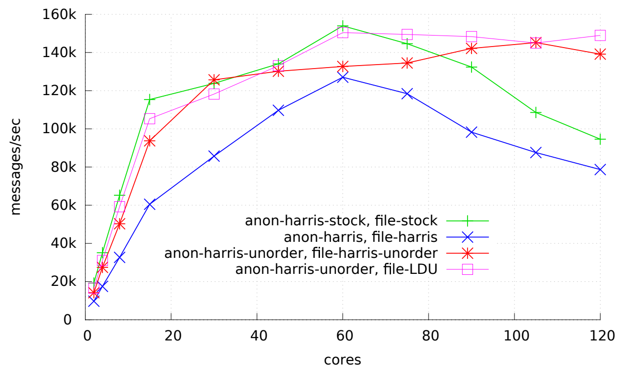

Figure 8: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

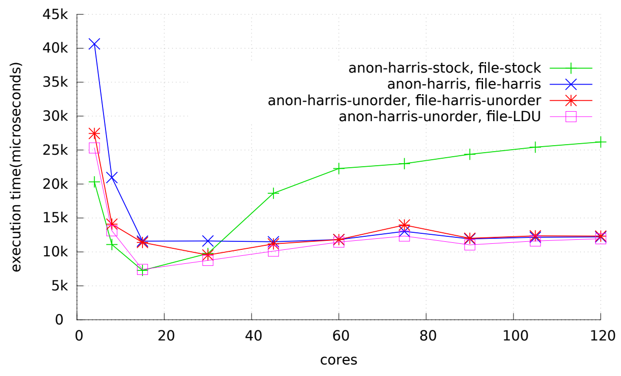

Figure 9: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

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.

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 [**?**]; we used 100 processes.

The results for lmbench are shown in Figure 9, 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 stock Linux waits to acquire reverse mapping locks such as `anon_vma's rwsem` and `mapping's i_mmap_rwsem`.

7 Discussion and future work

8 Related work

In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks.

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 system calls and page faults. These operations are synchronized by a single per-process lock. BonsaiVM [?] solved this address space problem by using the RCU; RadixVM [?] 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 [?].

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 [?] 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 [?] 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.

One method for the concurrent update is using the non-blocking algorithms [?] [?] [?], 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. These algorithms execute optimistically as though they read the value at location in their data structure;they may obtain stale data at the time. When they observed against the current value, they execute a CAS to compare the against value. The CAS fails when the value has been overridden, and they must be retried later on. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [?]. Moreover, none of the non-blocking algorithms implements an iterator, whose data structure just consists of the insert, delete and contains operations [?]. 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 [?] 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 [?] 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.

MCS [?], a scalable exclusive lock, is used in the Linux kernel [?]. 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 [?] 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 [?] 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.

9 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 opensource from https://github.com/KMU-embedded/scalablelinux.

10 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")